Improving the accuracy of dynamic branch prediction using branch correlation

Full text Pdf (918 KB)

Source **Architectural Support for Programming Languages and Operating Systems** archiv

Proceedings of the fifth international conference on Architectural support for pr and operating systems table of contents

Boston, Massachusetts, United States
Pages: 76 - 84
Year of Publication: 1992
ISSN:0362-1340
Also published in ...

Authors Shien-Tai Pan
Kimming So
Joseph T. Rahmeh

Sponsors SIGPLAN: ACM Special Interest Group on Programming Languages
IEEE : Institute of Electrical and Electronics Engineers
SIGOPS: ACM Special Interest Group on Operating Systems
SIGARCH: ACM Special Interest Group on Computer Architecture

Publisher ACM Press New York, NY, USA

DOI Bookmark: Use this link to bookmark this Article: http://doi.acm.org/10.1145/143365.1434
What is a DOI?

REFERENCES

Note: OCR errors may be found in this Reference List extracted from the full text art

expose the complete List rather than only correct and linked references.

1 A. Bashteen, I. Lui, J. Mullan, "A Superpipeline Approach to the MIPS Architecture IEEE Compcon'91, February 1991, pp. 8-12.

2 G.F. Grohoski, Machine organization of the IBM RISC System/6000 processor, IB and Development, v.34 n.1, p.37-58, Jan. 1990

3 W. W. Hwu , T. M. Conte , P. P. Chang, Comparing software and hardware schem of branches, Proceedings of the 16th annual international symposium on Computer a April 1989, Jerusalem, Israel

4 J.K.F. Lee, A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer De 17, 1, January, 1984, pp. 6-22.

5 S. McFarling , J. Hennesey, Reducing the cost of branches, Proceedings of the 13 symposium on Computer architecture, p.396-403, June 02-05, 1986, Tokyo, Japan

6 S. T Pan, K. So, J. T. Rahmeh, "Correlation-Based Branch Prediction," Technical F UT-CERC-TR-JTR91--O1, University of Texas at Austin, August, 1991.

7 James E. Smith, A study of branch prediction strategies, Proceedings of the 8th a Computer Architecture, p.135-148, May 12-14, 1981, Minneapolis, Minnesota, Unite

8 David W. Wall, Limits of instruction-level parallelism, Proceedings of the fourth in on Architectural support for programming languages and operating systems, p.176-: Santa Clara, California, United States

9 Workstation Performance, The SPEC Benchmark Suit Release 1.0, System Perfor Cooperative, June, 1990.

10 Tse-Yu Yeh , Yale N. Patt, Two-level adaptive training branch prediction, Procee international symposium on Microarchitecture, p.51-61, September 1991, Albuquerq Rico

11 T. Yoshida, T. Shimizu, S. Mizugaki, J. Hinata, 'The Gmicro/100 32-Bit Micropro August, 1991, pp. 20-23 & 62-72.

↑ **CITINGS** ***50***

Tse-Yu Yeh , Yale N. Patt, A comparison of dynamic branch predictors that use two l ACM SIGARCH Computer Architecture News, v.21 n.2, p.257-266, May 1993

Sangwook P. Kim , Gary S. Tyson, Analyzing the working set characteristics of branc Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitect 1998, Dallas, Texas, United States

Eric Hao , Po-Yung Chang , Yale N. Patt, The effect of speculatively updating branch

prediction accuracy, revisited, Proceedings of the 27th annual international symposiu p.228-232, November 30-December 02, 1994, San Jose, California, United States

Peter Petrov , Alex Orailoglu, Speeding up control-dominated applications through m customizations in embedded processors, Proceedings of the 38th conference on Desi p.512-517, June 2001, Las Vegas, Nevada, United States

Marius Evers , Sanjay J. Patel , Robert S. Chappell , Yale N. Patt, An analysis of corre predictability: what makes two-level branch predictors work, ACM SIGARCH Comput v.26 n.3, p.52-61, June 1998

S. Reches , S. Weiss, Implementation and analysis of path history in dynamic branch Proceedings of the 11th international conference on Supercomputing, p.285-292, Jul Austria

G. Palermo , M. Sam , C. Silvan , V. Zaccari , R. Zafalo, Branch prediction techniques processors, Proceedings of the 13th ACM Great Lakes Symposium on VLSI, April 28- D. C., USA

Tse-Yu Yeh , Yale N. Patt, Retrospective: alternative implementations of two-level ad prediction, 25 years of the international symposia on Computer architecture (selecte June 27-July 02, 1998, Barcelona, Spain

Erik Jacobsen , Eric Rotenberg , J. E. Smith, Assigning confidence to conditional brar Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitect December 02-04, 1996, Paris, France

Ravi Nair, Dynamic path-based branch correlation, Proceedings of the 28th annual ir on Microarchitecture, p.15-23, November 29-December 01, 1995, Ann Arbor, Michig

Kevin B. Theobald , Guang R. Gao , Laurie J. Hendren, Speculative execution and bra parallel machines, Proceedings of the 7th international conference on Supercomputir 1993, Tokyo, Japan

Dennis Lee , Jean-Loup Baer , Brad Calder , Dirk Grunwald, Instruction cache fetch p execution, ACM SIGARCH Computer Architecture News, v.23 n.2, p.357-367, May 19

A. R. Talcott , W. Yamamoto , M. J. Serrano , R. C. Wood , M. Nemirovsky, The impa branches on branch prediction scheme performance, ACM SIGARCH Computer Archit p.12-21, April 1994

James O. Bondi , Ashwini K. Nanda , Simonjit Dutta, Integrating a misprediction rec superscalar pipeline, Proceedings of the 29th annual ACM/IEEE international sympos Microarchitecture, p.14-23, December 02-04, 1996, Paris, France

Quinn Jacobson , Eric Rotenberg , James E. Smith, Path-based next trace prediction, annual ACM/IEEE international symposium on Microarchitecture, p.14-23, December Triangle Park, North Carolina, United States

Andreas Krall, Improving semi-static branch prediction by code replication, Proceedi

'94 conference on Programming language design and implementation, p.97-106, Jur Florida, United States

Tse-Yu Yeh , Deborah T. Marr , Yale N. Patt, Increasing the instruction fetch rate via prediction and a branch address cache, Proceedings of the 7th international conferer p.67-76, July 19-23, 1993, Tokyo, Japan

Todd C. Mowry , Chi-Keung Luk, Predicting data cache misses in non-numeric applic correlation profiling, Proceedings of the 30th annual ACM/IEEE international symposi Microarchitecture, p.314-320, December 01-03, 1997, Research Triangle Park, Nortl

Ching-Long Su , Alvin M. Despain, Minimizing branch misprediction penalties for sup Proceedings of the 27th annual international symposium on Microarchitecture, p.138 30-December 02, 1994, San Jose, California, United States

Barry Fagin , Kathryn Russell, Partial resolution in branch target buffers, Proceeding international symposium on Microarchitecture, p.193-198, November 29-December Michigan, United States

Gary Tyson , Matthew Farrens , John Matthews , Andrew R. Pleszkun, A modified ap management, Proceedings of the 28th annual international symposium on Microarch November 29-December 01, 1995, Ann Arbor, Michigan, United States

Toni Juan , Sanji Sanjeevan , Juan J. Navarro, Dynamic history-length fitting: a thirc branch prediction, ACM SIGARCH Computer Architecture News, v.26 n.3, p.155-166

Pierre Michaud , André Seznec , Richard Uhlig, Trading conflict and capacity aliasing predictors, ACM SIGARCH Computer Architecture News, v.25 n.2, p.292-303, May 1

Po-Ying Chang , Eric Hao , Yale N. Patt, Alternative implementations of hybrid branc Proceedings of the 28th annual international symposium on Microarchitecture, p.252 29-December 01, 1995, Ann Arbor, Michigan, United States

Brad Calder , Dirk Grunwald , Joel Emer, A system level perspective on branch archi Proceedings of the 28th annual international symposium on Microarchitecture, p.199 29-December 01, 1995, Ann Arbor, Michigan, United States

Joel Emer , Nikolas Gloy, A language for describing predictors and its application to a ACM SIGARCH Computer Architecture News, v.25 n.2, p.304-314, May 1997

Eric Rotenberg , Steve Bennett , James E. Smith, Trace cache: a low latency approa instruction fetching, Proceedings of the 29th annual ACM/IEEE international symposi Microarchitecture, p.24-35, December 02-04, 1996, Paris, France

Cliff Young , Nicolas Gloy , Michael D. Smith, A comparative analysis of schemes for prediction, ACM SIGARCH Computer Architecture News, v.23 n.2, p.276-286, May 1

Chih-Chieh Lee , I-Cheng K. Chen , Trevor N. Mudge, The bi-mode branch predictor, annual ACM/IEEE international symposium on Microarchitecture, p.4-13, December Triangle Park, North Carolina, United States

Kai Wang , Manoj Franklin, Highly accurate data value prediction using hybrid predic 30th annual ACM/IEEE international symposium on Microarchitecture, p.281-290, De Research Triangle Park, North Carolina, United States

Brad Calder , Dirk Grunwald, Next cache line and set prediction, ACM SIGARCH Com News, v.23 n.2, p.287-296, May 1995

T. K. Tan , A. K. Raghunathan , G. Lakishminarayana , N. K. Jha, High-level software macro-modeling, Proceedings of the 38th conference on Design automation, p.605-6 Vegas, Nevada, United States

Stuart Sechrest , Chih-Chieh Lee , Trevor Mudge, Correlation and aliasing in dynami SIGARCH Computer Architecture News, v.24 n.2, p.22-32, May 1996

Bryan Black , Bohuslav Rychlik , John Paul Shen, The block-based trace cache, ACM Architecture News, v.27 n.2, p.196-207, May 1999

Cliff Young , Michael D. Smith, Better global scheduling using path profiles, Proceedi ACM/IEEE international symposium on Microarchitecture, p.115-123, November 199 States

Scott Mahlke , Balas Natarajan, Compiler synthesized dynamic branch prediction, Pr annual ACM/IEEE international symposium on Microarchitecture, p.153-164, Decemt France

Evelyn Duesterwald , Vasanth Bala, Software profiling for hot path prediction: less is Operating Systems Review, v.34 n.5, p.202-211, Dec. 2000

B. Calder , D. Grunwald, Fast and accurate instruction fetch and branch prediction, A Architecture News, v.22 n.2, p.2-11, April 1994

Evelyn Duesterwald , Vasanth Bala, Software profiling for hot path prediction: less is Notices, v.35 n.11, p.202-211, Nov. 2000

Eitan Federovsky , Meir Feder , Sholomo Weiss, Branch prediction based on universa algorithms, ACM SIGARCH Computer Architecture News, v.26 n.3, p.62-72, June 19

Nicolas Gloy , Cliff Young , J. Bradley Chen , Michael D. Smith, An analysis of dynam schemes on system workloads, ACM SIGARCH Computer Architecture News, v.24 n.

I-Cheng K. Chen , John T. Coffey , Trevor N. Mudge, Analysis of branch prediction vi ACM SIGPLAN Notices, v.31 n.9, p.128-137, Sept. 1996

Thomas Ball , Peter Mataga , Mooly Sagiv, Edge profiling versus path profiling: the s of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languag 19-21, 1998, San Diego, California, United States

Dionisios N. Pnevmatikatos , Manoj Franklin , Gurindar S. Sohi, Control flow predicti processors, Proceedings of the 26th annual international symposium on Microarchite

December 01-03, 1993, Austin, Texas, United States

Cliff Young , Michael D. Smith, Improving the accuracy of static branch prediction us ACM SIGPLAN Notices, v.29 n.11, p.232-241, Nov. 1994

Brad Calder , Dirk Grunwald, Reducing branch costs via branch alignment, ACM SIGI p.242-251, Nov. 1994

Brad Calder , Dirk Grunwald, Reducing indirect function call overhead in C++ progra 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, p.3 1994, Portland, Oregon, United States

Cliff Young , Michael D. Smith, Static correlated branch prediction, ACM Transactions Languages and Systems (TOPLAS), v.21 n.5, p.1028-1075, Sept. 1999

Nicolas Gloy , Michael D. Smith , Cliff Young, Performance issues in correlated branc Proceedings of the 28th annual international symposium on Microarchitecture, p.3-1 29-December 01, 1995, Ann Arbor, Michigan, United States

Gabriel H. Loh, Exploiting data-width locality to increase superscalar execution band the 35th annual ACM/IEEE international symposium on Microarchitecture, November Turkey

↑ **INDEX TERMS**

Primary Classification:
C. Computer Systems Organization
↳ C.1 PROCESSOR ARCHITECTURES

Additional Classification:
C. Computer Systems Organization

General Terms:
Design, Languages, Measurement, Performance

↑ **Collaborative Colleagues of:**

Shien-Tai Pan:

Joseph T. Rahmeh
Kimming So

Joseph T. Rahmeh:

Jacob A. Abraham
Edward S. Davidson
Peter Y. T. Hsu
Sankaran Karthik
Shien-Tai Pan
Lawrence T. Pillage
Kimming So
Indira de Souza
Dah-Cherng Yuan

Kimming So:

E. G. Coffman
Tse-Yun Feng
Shien-Tai Pan
Joseph T. Rahmeh
Rudolph N. Rechtschaffen
Herbert D. Schwetman
Chuan-lin Wu

This Article has also been published in:

ACM SIGPLAN Notices
Volume 27 , Issue 9 (September 1992)

Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation

Shien–Tai Pan
Advanced Workstation Division
IBM Corporation, 9440
11400 Burnet Road
Austin, Texas 78758–3493

Kimming So
Advanced Workstation Division
IBM Corporation, 2808
11400 Burnet Road
Austin, Texas 78758–3493
sok@watson.ibm.com

Joseph T. Rahmeh
Electrical and Computer
Engineering Department
University of Texas at Austin
Austin, Texas 78712
rahmeh@cerc.austin.edu

Abstract

Long branch delay is a well–known problem in today's high performance superscalar and superpipeline processor designs. A common technique used to alleviate this problem is to predict the direction of branches during the instruction fetch. Counter–based branch prediction, in particular, has been reported as an effective scheme for predicting the direction of branches. However, its accuracy is generally limited by branches whose future behavior is also dependent upon the history of other branches. To enhance branch prediction accuracy with a minimum increase in hardware cost, we propose a correlation–based scheme and show how the prediction accuracy can be improved by incorporating information, not only from the history of a specific branch, but also from the history of other branches. Specifically, we use the information provided by a proper subhistory of a branch to predict the outcome of that branch. The proper subhistory is selected based on the outcomes of the most recently executed M branches. The new scheme is evaluated using traces collected from running the SPEC benchmark suite on an IBM RISC System/6000 workstation. The results show that, as compared with the 2–bit counter–based prediction scheme, the correlation–based branch prediction achieves up to 11% additional accuracy at the extra hardware cost of one shift register. The results also show that the accuracy of the new scheme surpasses that of the counter–based branch prediction at saturation.

1. Introduction

Recent advances in RISC architectures and VLSI technologies allow computer designers to exploit more instruction–level parallelism with deeper pipelines and more concurrent functional units [1, 2]. As sophisticated processors are built to exploit the available instruction–level parallelism, more attention needs to be paid to the disruption of pipeline flow as a result of branch instruction execution [8]. Pipeline disruption reduces the effective instruction throughput by introducing extra delays in the pipeline. Since branches constitute a large portion of all the executed instructions, the efficiency of handling branches is important. Our primary interest is in reducing the branch penalty incurred in executing conditional branches. All branches mentioned below, unless otherwise stated, are conditional branches.

Almost all the branch cost reduction techniques reported in the literature require the use of some mechanism for predicting the outcome of branches. Other than the profiling technique [3, 5], all prediction schemes require hardware assistance. Hardware–assisted branch predictions typically fall into two categories: *static* and *dynamic*. Overview of these schemes can be found in [4, 7]. Generally, dynamic prediction gives better results than static prediction, but at the cost of increased hardware complexity. A less–complex yet reasonably effective scheme is the ***N–bit counter scheme*** [3, 4, 7]. In this scheme, the prediction of the outcome of a branch is based on the output of a finite–state machine whose state is recorded in an N–bit up/down counter. The counter is incremented or decremented according to whether the branch is taken or not. We refer to this scheme as the ***counter–based*** branch prediction. Later we will show its operation in more detail.

A common limitation with most of the dynamic branch prediction schemes is that the prediction is based on "*self–history*". Specifically, the prediction is based exclusively on the past history of the branch under consideration, completely ignoring the information provided by the executions of other branches. Self–history prediction schemes generally work well for scientific/engineering applications where program execution is dominated by inner–loops. However, in many integer workloads, control–flows are complex and very often the outcome of a branch is affected by the outcomes of recently executed branches. In other words, the branches are **correlated**. Because of the correlation, the history of a branch, considered by itself, is very chaotic and that reduces the accuracy of self–history

ASPLOS V - 10/92/MA,USA
© 1992 ACM 0-89791-535-6/92/0010/0076...$1.50

prediction schemes. A prior study shows that *branch correlation* does take place in programs and that its source can be traced back to common high–level language constructs [6]. The Appendix summarizes some of our observations of the source code–level branch correlation that appear in the SPEC integer benchmarks.

Contrary to the self–history based approach, the *"two–level adaptive training branch prediction"* reported recently uses the global branch history pattern associated with each branch address for predicting the outcome of the branch [10]. The same global history pattern results in the same prediction, regardless of which branch address the history pattern is associated with. Although this approach is reported to produce a fairly high prediction accuracy [10], its hardware implementation seems quite complicated.

To our knowledge, very little work has been done in addressing the issue of branch correlation in branch prediction. In this paper, we study the effect of branch correlation in branch prediction and propose a **correlation–based** prediction scheme which also produces high prediction accuracy. The proposed branch prediction scheme is simple to implement and its implementation is very similar to that of the counter–based branch prediction.

The new scheme is evaluated using traces collected from running the SPEC benchmark suite [9] on an IBM RISC System/6000 workstation. The results show that, as compared with the 2–bit counter–based prediction scheme, the correlation–based branch prediction achieves up to 11% additional accuracy at the extra hardware cost of one shift register. The results also show that the accuracy of the new scheme surpasses that of the counter–based branch prediction at saturation.

The remainder of this paper is organized as follows: In section 2, the correlation–based branch prediction scheme is introduced with an example. A brief description of the counter–based branch prediction is also given. In section 3, simulation results evaluating the new scheme are presented. In section 4, we give the main conclusions.

2. Dynamic Branch Prediction

In this section, we will describe the N–bit counter scheme and introduce a new prediction scheme based on branch correlation. An example will be given to explain the difference between these two schemes. Finally, the implementation of the new scheme will be discussed.

2.1 Counter–Based Branch Prediction

The basic idea for the counter–based branch prediction is to use an N–bit up/down counter [3, 4, 7] for prediction. In the ideal case, an N–bit counter (with some initial value) is assigned to each *static branch* (branches with distinct addresses). When a branch is about to be executed, the counter value C, associated with that branch, is used for prediction. If C is greater than or equal to a predetermined threshold value L, the branch is predicted taken, otherwise it is predicted not taken. A typical value for L is 2^{N-1}. The counter value C is updated whenever that branch is resolved. If the branch is taken, C is incremented by one, otherwise it is decremented by one. If C is 2^N-1, it remains at that value as long as the branch is taken. If C is 0, it remains at zero as long as the branch is not taken.

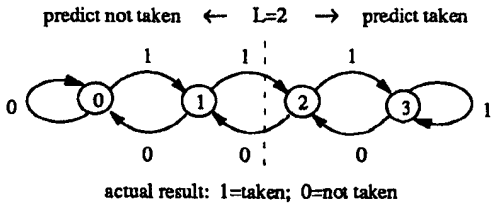


Fig. 1 FSM for the 2–bit Counter Scheme

The operation of the N–bit counter scheme corresponds to a finite–state machine (FSM) with 2^N states. Fig. 1 shows the FSM with N=2 and L=2. Smith [7] reported that a counter of 2 bits is usually as good or better than other strategies and a larger counter size does not necessarily give better results.

2.2 Correlation–Based Branch Prediction

Most studies of dynamic branch prediction focus on the history of the branch under consideration [4, 7]. With hardware–assisted branch prediction, only the most recent history of a branch is used to predict the outcome of that branch. These branch prediction schemes work well for scientific/engineering workloads where program execution is dominated by inner–loops. However, they do not work as well for integer workloads where the outcome of a branch is affected by the outcomes of recently executed branches. When one branch depends on another, in the sense that its outcome depends on the outcome of the other branch, we say that the branches are **correlated**.

As an illustration of branch correlation, consider the code fragment shown in Fig. 2:

```
if (aa==2)                    /* b1 */
      aa = 0;
if (bb==2)                    /* b2 */
      bb = 0;
if (aa != bb) {               /* b3 */
      .....
}
```

Fig. 2 A Code Fragment from SPEC Benchmark *eqntott*

This code fragment (other than the comments) appears in a frequently executed block of the SPEC integer benchmark *eqntott*. There are three *if*–statements in this code fragment. Assume that the *if*–statements are converted by a compiler to three branch instructions b_1, b_2, and b_3, and the action determined by each *if*–statement is the branch "fall–through path", meaning that the branch "taken" path is the path for which the condition is not true. Since the outcome of b_3 depends on the values of *aa* and *bb*, it is obvious that b_3 is **correlated** with b_1 and b_2.

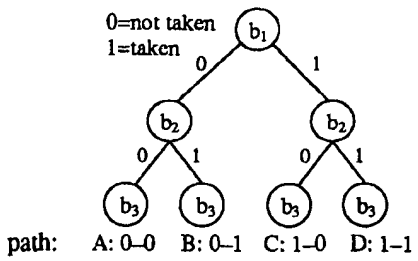


Fig. 3 Branch Tree for the Code Fragment Given in Fig. 2

Although the presence of branch correlation may cause the behavior of a branch to appear more random, it may shed some light on the condition upon which the branch decision is based. Consider again the same example given in Fig. 2. After the executions of b_1 and b_2, the condition that b_3 is dependent upon is already partially known. Fig. 3 shows the part of the branch tree before the execution of b_3 given that b_1 and b_2 have been executed.

There are four possible paths reaching b_3 through the executions of b_1 and b_2. For example, if b_1 is taken and b_2 is not taken, then b_3 is reached via the 1–0 path (path C in Fig. 3). Fig. 4 shows the information available at b_3, given that b_1 and b_2 have been executed. It is clear that if b_3 is reached via the 0–0 path, the outcome of b_3 can be determined prior to its execution. But this situation cannot be exploited by the conventional self–history based prediction schemes. This example suggests that the outcome of a branch can be more readily determined if the path leading to it is known. By splitting the branch history of b_3 into four **subhistories** according to the paths leading to b_3, one may reduce the randomness of the apparent behavior of b_3 and thus make a better prediction.

path leading to b_3:	A: 0–0	B: 0–1	C: 1–0	D: 1–1
	$\begin{cases} aa=0 \\ bb=0 \end{cases}$	$\begin{cases} aa=0 \\ bb\neq 2 \end{cases}$	$\begin{cases} aa\neq 2 \\ bb=0 \end{cases}$	$\begin{cases} aa\neq 2 \\ bb\neq 2 \end{cases}$

Fig. 4 Information About *aa*, *bb* Available at b_3 After b_1 and b_2 Have Been Executed

Let's further examine the example with data that are arbitrarily chosen only to reflect the branch correlation. Suppose that we run the code fragment given in Fig. 2 on a machine which implements the 2–bit counter scheme shown in Fig. 1 with initial state set to 0. Table 1 shows the predicted outcomes of b_3 and the state transitions. The first two columns show the initial values of *aa* and *bb* before the execution of b_1. Columns *aa'* and *bb'* in the table show the new values of *aa* and *bb* after b_1 and b_2 are executed. Column "*path*" indicates the path from which b_3 is reached. Column "*curr state*" shows the current state of the FSM. Column "*pred*" shows the predicted outcome of b_3. The actual outcome is given in column "*act*". Column "*c/w*" indicates the correct (c) or wrong (w) prediction. The state is updated according to the current state and the actual outcome. The updated state is shown in the column under "*next state*".

Table 1 State Transitions and Branch Predictions for b_3 Using 2–bit Counter–Based Prediction Scheme

aa	*bb*	*aa'*	*bb'*	*path*	*curr state*	*pred*	*act*	*c/w*	*next state*
0	2	0	0	C	0	N	T	w	1
2	2	0	0	A	1	N	T	w	2
2	1	0	1	B	2	T	N	w	1
2	0	0	0	B	1	N	T	w	2
2	2	0	0	A	2	T	T	c	3
1	0	1	0	D	3	T	N	w	2
1	0	1	0	D	2	T	N	w	1
2	0	0	0	B	1	N	T	w	2
0	1	0	1	D	2	T	N	w	1
1	1	1	1	D	1	N	T	w	2
1	2	1	0	C	2	T	N	w	1
1	2	1	0	C	1	N	N	c	0
2	2	0	0	A	0	N	T	w	1
2	0	0	0	B	1	N	T	w	2
0	1	0	1	D	2	T	N	w	1
2	2	0	0	A	1	N	T	w	2
0	2	0	0	C	2	T	T	c	3
0	1	0	1	D	3	T	N	w	2
1	0	1	0	D	2	T	N	w	1
2	2	0	0	A	1	N	T	w	2

N=not taken, T=taken, c=correct pred., w=wrong pred.

A careful inspection of the table reveals that the apparently random branch history of b_3 (column "*act*") is actually formed by interweaving four less random branch subhistories, each of which is associated with a branch path leading to b_3 (compare columns "*path*" and "*act*"). After splitting the branch history of b_3 according to the four branch paths shown in Fig. 5, one can obtain the four branch subhistories of b_3.

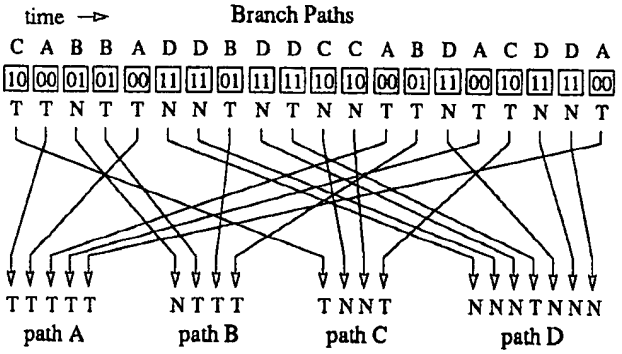


Fig. 5 Subhistories Obtained by Splitting the History of b_3 According to the Branch Paths

It is evident from Fig. 5 that the outcomes of b_3 are less random within each subhistory. Hence better predictions are expected if we independently implement a 2–bit counter for each subhistory. In fact, only 3 out of the 20 executions of b_3 are correctly predicted if only one 2–bit counter (with initial state equal to 0) is used. However, if four 2–bit counters are used (all initialized to 0), with one for each subhistory, 10 additional correct predictions can be obtained. Note that the state transition and the state update of the FSM associated with each counter are local to each branch path. This is shown in Fig.

6. Notice that we are not suggesting to use four counters for "each" branch. We are merely showing that taking the path leading to a branch into consideration leads to a better prediction. Later we will show an implementation scheme that exploits the "correlation" between branches without increasing the overall number of counters used to track the history of branches.

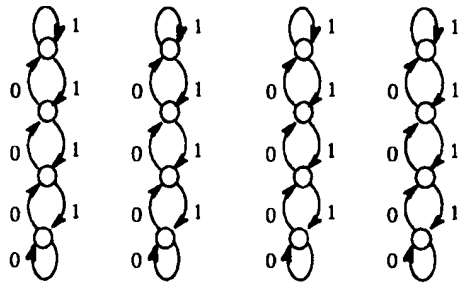


Fig. 6 FSMs using Four 2–bit Counters

Fig. 6 suggests that in order to select the proper 2–bit counter assigned to each subhistory for prediction, one needs to memorize the branch path leading to b_3. This can be achieved by using a 2–bit shift register which records the outcomes of the two most recently executed branches. The shift register is then used to select the appropriate counter. The use of a shift register for tracking and selectively relating the correlated information to proper branch subhistory is the main idea of the proposed **correlation–based** branch prediction. Basically, the proposed scheme uses the branch path information to split the history of a branch into several subhistories and selectively use the proper subhistory information for predicting the outcome of the branch.

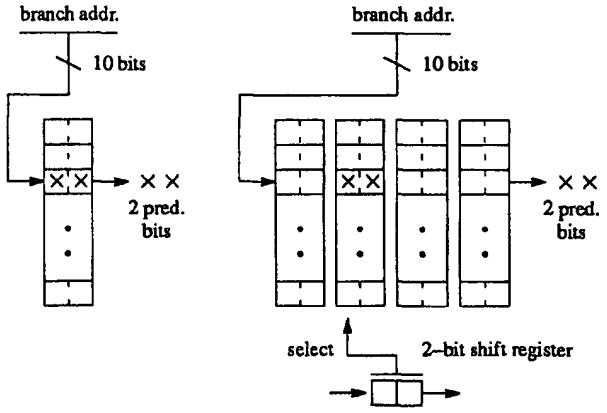
Generally, an **M–step** correlation–based branch prediction uses the outcomes of the last M branches (including unconditional branches) seen by the machine to split the history of a branch into 2^M subhistories. The prediction is then done independently within each subhistory using any (or the best) history–based branch prediction algorithm. A good candidate for prediction within each subhistory is the N–bit counter–based branch prediction mentioned earlier. In this case, an **M–bit shift register** is required to store the outcomes of the last M branch executions (0 for not taken, 1 for taken). This shift register is able to identify a total of 2^M subhistories of a branch. Within each subhistory, the prediction is done using an N–bit counter associated with it. There are a total of 2^M FSM's associated with each branch. Everytime the outcome of a branch is to be predicted, the M–bit shift register is used to select the proper FSM, resulting in a set of N prediction bits. Once the FSM is selected, the prediction and the state update are done according to the N–bit counter–based prediction algorithm.

In the following, we will refer to this scheme as the **(M,N) correlation–based** branch prediction scheme or simply the **(M,N) correlation scheme**, meaning that an M–bit shift register is used to select an N–bit counter for prediction. The **number of correlation steps** is defined as the number of bits in the shift register. When the prediction scheme used within each subhistory is understandable without any ambiguity, we will simply refer to it as an **M–step correlation scheme.**

2.3 Implementation

When the N–bit counter scheme or the (M,N) correlation scheme is implemented by itself, a table is required to store the prediction information. We refer to this table as the "*branch prediction table*" or briefly, BPT. Fig. 7 (a) shows the logical organization of a 1K–entry BPT for the 2–bit counter scheme, with each entry containing 2 prediction bits. Fig. 7 (b) shows the logical organization of a 1K–entry BPT for the (2,2) correlation scheme, with each entry containing $2\times2^2=8$ prediction bits.

Notice the difference in physical size of the two tables, even though the number of logical entries is identical. In general, if a 2^I–entry table is used for (M,N) correlation scheme, a total of $N\times2^{I+M}$ bits is required for the table, with each entry containing 2^M sets of N prediction bits. The table is generally accessed using the low–order I bits of the branch address. However, depending on the implementation, the table may be accessed using the address of the instruction immediately prior to the branch under consideration [11]. Once the entry is determined, the M–bit shift register which stores the outcomes of the last M branches is used to select the proper set of the N bits from the entry. These N bits are used for predicting the outcomes of all branches whose addresses are mapped into the same entry.

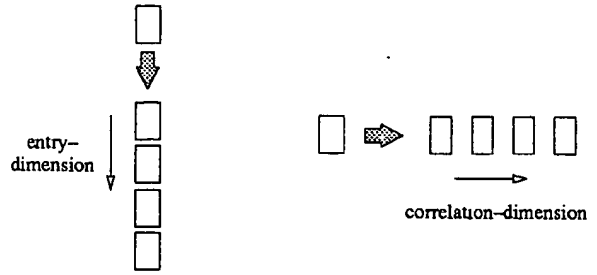


(a) 2–bit counter scheme (b) (2,2) correlation scheme

Fig. 7 Logical Organization of a 1K–Entry Table

A design tradeoff in implementing the dynamic branch prediction usually involves in choosing the physical size of the BPT for a desired prediction accuracy. It is interesting to note from Fig. 7 that if the BPT size is to be changed, two "logical directions" can be considered. The table size can be increased/decreased either along the vertical direction as shown in Fig. 8 (a) or along the horizontal direction as shown in Fig. 8 (b). Fig. 8 (a) is typical for implementing the

counter–based scheme whereas Fig. 8 (b) is typical for correlation schemes. We will refer to the directions shown in Fig. 8 (a) and (b) as the **entry–dimension** and the **correlation–dimension**, respectively. Of course, any combination of the two is possible.

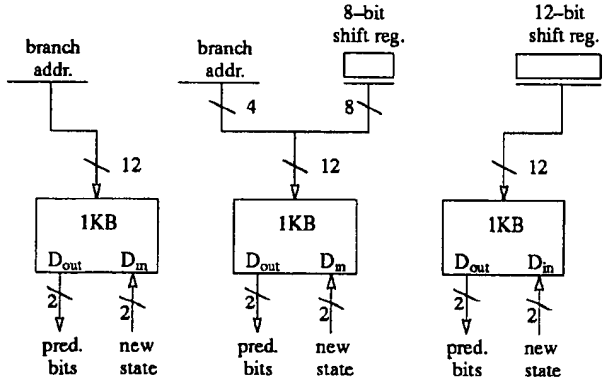


(a) increased along the entry–dimension

(b) increased along the correlation–dimension

Fig. 8 Increasing the Size of a BPT

While the logical organization and the behavior of the tables for the counter and the correlation schemes are different, the physical implementations are quite similar. Fig. 9 shows the implementation using a 1KB–BPT. When this table is used for the 2–bit counter scheme, 12 bits are required for a table lookup (Fig. 9 (a)). As mentioned earlier, these 12 bits are usually obtained from the branch address. However, if the same table is used for correlation schemes, some of the bits for table lookup are obtained from the shift–register. For example, if the (8,2) correlation scheme is implemented as shown in Fig. 8 (b), the bits for table lookup consist of 8 bits from the shift register and 4 bits from the branch address. It is important to note that as a correlation scheme is implemented instead of the original 2–bit counter scheme using the same size of table, the only extra hardware cost incurred by the correlation scheme is the shift register (Fig. 9 (b)).



(a) 2–bit counter scheme

(b) (8,2) correlation scheme

(c) (12,2) correlation scheme – degenerate

Fig. 9 Physical Implementation Using a 1KB–BPT

Fig. 9 (b) also shows an interesting case: as the table size is fixed, the larger the shift register used, the fewer branch address bits are required. In other words, as the table size is fixed, increasing the size of the shift register is equivalent to "squashing" the BPT along the entry–dimension. An interesting extreme case occurs when the table degenerates to a single–entry table. In this case, the bits for table lookup are obtained entirely from the shift register. Fig. 9 (c) shows the degenerate case for a 1KB–BPT. This case is equivalent to implementing the (12,2) correlation scheme using a single–entry table shown in Fig. 10. Similarly, Fig. 9 (a) can be thought as the other extreme case when the table in Fig. 9 (b) is "squashed" along the correlation–dimension. The advantage of considering the degenerate case is that its table lookup depends only on the shift register, completely independent of the branch address. Because of this unique characteristic, a resolved branch always predicts the outcome of the next branch. The degenerate case of the correlation scheme is interesting, not only because of its simple implementation, but also because the predicted outcome of a branch can be known way before the execution of that branch.

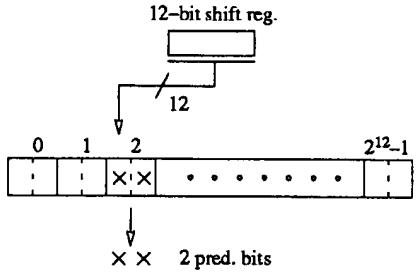


Fig. 10 Degenerate Case for a 1KB–BPT

3. Trace–Driven Simulations & Results

Trace–driven simulations are used to examine the (M,2) correlation schemes for BPT's with entries ranging from 1 to 32K. Due to the limitation of the program size and simulation time, only $M \leq 10$ are evaluated for non–degenerate cases and $M \leq 15$ for degenerate cases. Note that the scheme (0,2) corresponds to the original 2–bit counter scheme. The programs used for the experiment are from the SPEC benchmark suite. The traces are collected using a trace program and commercially available C and FORTRAN compilers for the IBM RISC System/6000 system. Table 2 summarizes the trace lengths and branch statistics for the benchmarks used in this study. The **accuracy**, defined as the *percentage of correct predictions*, will be used as the metric for measuring the efficiency of branch prediction.

For SPEC floating–point benchmarks *nasa7*, *matrix300*, and *tomcatv*, no difference is found between correlation–based schemes and the 2–bit counter scheme. All predict with more than 99% accuracy. These results are not surprising for loop–intensive scientific/engineering applications where programming structures are dominated by simple loops. Because of this, only the results of the other 7 SPEC benchmarks, namely, *doduc*, *spice*, *fpppp*, *gcc*, *espresso*, *eqntott*, and *li*, are presented. For convenience, we will use the short-

hand "7 SPEC benchmarks" or "7 benchmarks" to mean these 7 benchmarks, "floating-point benchmarks" to mean the benchmarks *doduc, spice,* and *fpppp,* and "integer benchmarks" to mean the benchmarks *gcc, espresso, eqntott,* and *li.*

Table 2 Branch Statistics for SPEC Benchmarks

	Inst.	b_u	b_c	p	q	s
spice	50M	.093	.125	.538	.196	41.3
doduc	50M	.020	.094	.630	.551	137.2
nasa7	50M	~0	.166	.994	.993	0.6
matrix300	50M	.001	.198	.993	.993	1.7
fpppp	50M	.005	.016	.575	.450	197.2
tomcatv	50M	~0	.059	.993	.993	72.6
gcc	50M	.041	.189	.635	.556	800.3
espresso	50M	.071	.193	.538	.369	46.7
li	50M	.062	.165	.601	.450	39.7
eqntott	50M	.021	.305	.445	.406	2.8

b_u: frequency of unconditional branches
b_c: frequency of conditional branches
p: probability that a branch is taken
q: probability that a conditional branch is taken
s: static conditional branches per 1million executed conditional branches

3.1 Accuracy for Fixed Table Size

We first compare the correlation-based scheme with the 2-bit counter scheme using the same 1KB-BPT. Notice that the number of table entries for the two schemes are different (see Fig. 7). A 1KB-table has 4K entries when the 2-bit counter scheme is implemented, whereas the same table has only 16 entries when the (8, 2) correlation schemes is implemented.

Fig. 11 shows the results for a 1KB-BPT. The figure compares the accuracy obtained by implementing the 2-bit counter scheme and the additional accuracy gained by implementing the (8, 2) correlation scheme. Since the 2-bit counter scheme has already provided very high accuracies for *doduc* and *espresso* (about 95%), there is very little chance for correlation schemes to gain more accuracy. The benchmark *gcc* shows very little improvement in accuracy. This is because that a 1KB-table is not large enough to contain most of the frequently executed branches in *gcc.*

The remaining benchmarks show considerable improvements in accuracy. The two biggest gains in accuracy are obtained by *eqntott* and *li.* Since branches in *eqntott* are highly correlated, the 2-bit counter scheme cannot provide high accuracy (only about 83%). More than 11% of additional accuracy can be attained by the correlation scheme.

The second highest improvement in accuracy is achieved by *li* (more than 5%). It is known that *li* is a "pointer-chasing" oriented program where a compiler may generate *load, compare,* and *branch* instructions in sequence over and over again. The branch correlation exists wherever the data loaded for determining the branch direction is affected by the directions of prior branches. As reported in [2], a *compare-branch* pair of instructions in the IBM RISC System/6000 machine causes a 3-cycle bubble in the pipeline. The correlation-based scheme proposed here is particularly useful to reduce such delay.

Although we have only shown the results for the 8-step correlation scheme, it is observed from the simulation that, as the number of table entries is fixed, the accuracy increases as the number of correlation steps increases. This observation is true for all the 7 benchmarks.

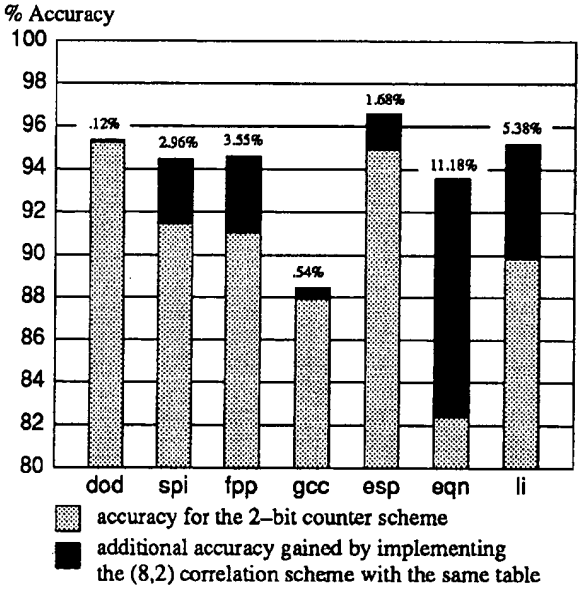


Fig. 11 Accuracies for an 1KB-BPT: (0,2) v.s. (8,2)

3.2 Accuracy at the Limiting Case

It is observed that the accuracy provided by the 2-bit counter scheme asymptotically approaches certain limit as the BPT size increases. Fig. 12 shows the limit at which the 2-bit counter scheme saturates. When the table is large enough to contain most of the frequently executed branches, the prediction capability of the 2-bit counter scheme reaches its inherent limits. As we mentioned earlier, one of the limitations of the 2-bit counter scheme is that it is self-history based. Since the correlation scheme provides better prediction by incorporating the information from other branches, it can surpass the limit at which the 2-bit counter scheme saturates.

As an illustration, consider the accuracy curves for *li* shown in Fig. 13. It is clear that the accuracy provided by the 2-bit counter scheme saturates at a table of 2K entries. Increasing the table size along the entry-dimension as shown in the figure makes very little improvement in accuracy. However, if the BPT size is increased along the correlation dimension (see Fig. 8 (b)), more accuracy can be gained. Fig. 12 shows the additional accuracy achievable by the correlation scheme for the 7 benchmarks.

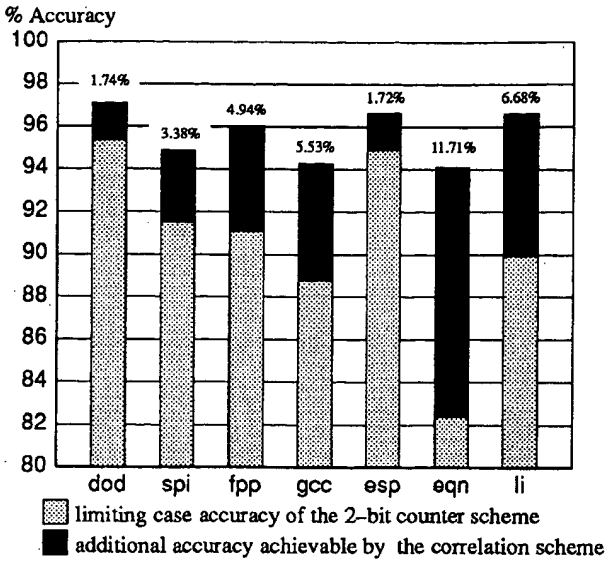


Fig. 12 Limiting Case Accuracy

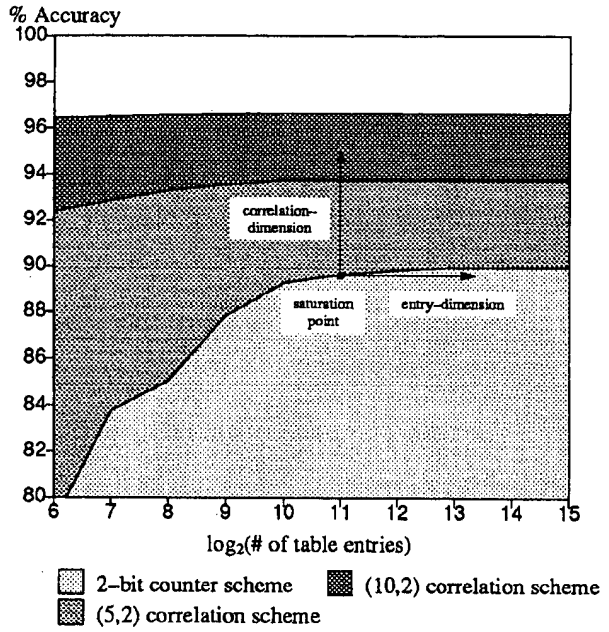


Fig. 13 Prediction Accuracy for *li*

3.3 Accuracy at the Degenerate Case

The degenerate correlation scheme provides an interesting case for a practical implementation, since its table lookup doesn't depend on the branch address. Because of this unique characteristic, the table lookup for the next branch can be done as soon as the current branch is resolved. This is attractive to timing-critical implementations of the branch prediction.

The only disadvantage with the degenerate case is that the table must be very large in order to outperform the 2-bit counter scheme. This is due to the fact that enormous amount of address conflicts are introduced with an one-entry table (Fig. 10). However, the effect of address conflict is attenuated when the table size is large. It is observed from the simulation that a larger correlation step is required before the degenerate case has a noticeable improvement over the 2-bit counter scheme. Table 3 summarizes the observation.

It is also observed that when the table size is large, the degenerate case sometimes performs better than the non-degenerate case. Fig. 14 shows the results of implementing the degenerate (15,2) scheme using an 8KB-table.

Table 3 # of Correlation Steps Required Before Degenerate Case has Noticeable Improvement Over the 2-Bit Counter Scheme

doduc	*spice*	*fpppp*	*gcc*	*espresso*	*eqntott*	*li*
15	6	10	14	8	5	11

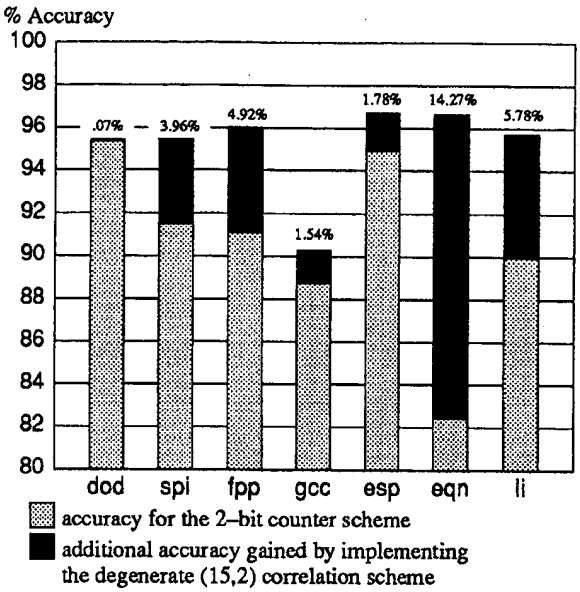


Fig. 14 Accuracy for an 8KB-BPT: (0,2) v.s. Degenerate (15,2)

4. Conclusions

In this paper, we have proposed a novel dynamic branch prediction scheme which uses the proper subhistory information of a branch to predict the outcome of that branch. The key idea is to relate the subhistory which is being selected to the most recently executed branches via a shift register. The new scheme is evaluated using traces collected from running the SPEC benchmark suite on an IBM RISC System/6000 machine. It is shown that the proposed new scheme gives considerably higher accuracy than that of the 2-bit counter prediction scheme at the extra hardware cost of one shift register. We have observed from the simulation that for the same BPT of size 1KB or above, the (M,2) correlation scheme generally provides the best improvement in accuracy over the 2-bit counter scheme for $5 \leq M \leq 8$. We want to emphasize that as more instruction-

level parallelism is exploited by today's superscalar and superpipelined processors, few percent increase in branch prediction accuracy is significant in improving the overall processor performance.

We have demonstrated that the new scheme is simple and easy to implement. It provides a new dimension as a design alternative for increasing the BPT size, i.e., the *correlation–dimension*. We have also shown that the accuracy of the correlation scheme surpasses that of the 2–bit counter scheme at saturation.

Acknowledgements

We would like to thank Ju–ho Tang of IBM T. J. Watson Research Center for providing the tracing tool, Chin–Cheng Kau, Ed Silha, Wade Shaw of IBM/Austin, and Kate Stewart of IBM/Toronto for reviewing our earlier drafts, and the management of IBM Advanced Workstation Division/Systems Architecture & Performance for their support of this research.

Reference

[1] A. Bashteen, I. Lui, J. Mullan, "A Superpipeline Approach to the MIPS Architecture," *Proceedings of the IEEE Compcon'91, February 1991, pp. 8–12.*

[2] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. of Research and Development, Vol. 34, No. 1, January 1990, pp. 37–58.*

[3] W. M. Hwu, T. M. Conte, P. P. Chang, "Comparing Software and Hardware Schemes for Reducing the Cost of Branches," *Proceedings of the 16th Annual International Symposium on Computer Architecture, May, 1989, pp. 224–233.*

[4] J. K. F. Lee, A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer, 17, 1, January, 1984, pp. 6–22.*

[5] S. McFarling, J. Hennessy, "Reducing the Cost of Branches," *Proceedings of the 13th Annual International Symposium on Computer Architecture, June, 1986, pp. 396–403.*

[6] S. T. Pan, K. So, J. T. Rahmeh, "Correlation–Based Branch Prediction," *Technical Report, UT–CERC–TR–JTR91–01, University of Texas at Austin, August, 1991.*

[7] J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Annual International Symposium on Computer Architecture, June, 1981, pp. 135–147.*

[8] D. W. Wall, "Limits of Instruction–Level Parallelism," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991, pp176–188.*

[9] Workstation Performance, The SPEC Benchmark Suit Release 1.0, *System Performance Evaluation Cooperative, June, 1990.*

[10] T. Y. Yeh, Y. N. Patt, "Two–Level Adaptive Training Branch Prediction," *Proceedings of the 24th Annual International Symposium on Microarchitecture, November, 1991, pp. 51–61.*

[11] T. Yoshida, T. Shimizu, S. Mizugaki, J. Hinata, "The Gmicro/100 32–Bit Microprocessor," *IEEE Micro, August, 1991, pp. 20–23 & 62–72.*

Appendix

Examples of Source Code–Level Branch Correlation from the SPEC Integer Benchmarks:

benchmark: eqntott file name: pterm_ops.c

```
if (aa == 2)
    aa = 0;
if (bb == 2)
    bb = 0;
if (aa != bb) {
    .....
}
```

benchmark: eqntott file name: pterm_ops.c

```
while (low <= high) {
    i = (high + low) / 2;
    if (H (i) < hsh)
        low = i + 1;
    else if (i > 0 && H (i–1) >= hsh)
        high = i – 1;
    else if (H (i) == hsh)
        break;
    else  return (NIL_PTERM);
}
```

benchmark: eqntott file name: ucbqsort.c

```
j = (j == jj ? i : jj);
if ((*qcmp)(j, tmp) < 0)
    j = tmp;
```

benchmark: li file name: xllist.c

```
while (*adstr && consp(list))
 list = (*adstr++ == 'a' ? car(list) : cdr(list));
```

benchmark: li file name: xlread.c

```
while ((ch = xlpeek(fptr)) != EOF) {
  if (islower(ch)) ch = toupper(ch);
  if (!isdigit(ch) && !(ch >= 'A' && ch <= 'F'))
    break;
    .....
}
```

benchmark: li file name: xlmath.c

```
if (imode)
  switch (fcn) {
  case '<':    icmp = (icmp < 0); break;
  case 'L':    icmp = (icmp <= 0); break;
  case '=':    icmp = (icmp == 0); break;
  case '#':    icmp = (icmp != 0); break;
  case 'G':    icmp = (icmp >= 0); break;
  case '>':    icmp = (icmp > 0); break;
  }
else
  switch (fcn) {
  case '<':    icmp = (fcmp < 0.0); break;
  case 'L':    icmp = (fcmp <= 0.0); break;
  case '=':    icmp = (fcmp == 0.0); break;
  case '#':    icmp = (fcmp != 0.0); break;
  case 'G':    icmp = (fcmp >= 0.0); break;
  case '>':    icmp = (fcmp > 0.0); break;
  }
return (icmp ? true : NIL);
```

benchmark: li file name: xlcont.c

```
rbreak = FALSE;
while (xleval(test) == NIL) {
  if (tagblock(arg,&rval)) {
    rbreak = TRUE;
    break;
  }
}
if (!rbreak)
  .....
```

benchmark: espresso file name: compl.c

```
for(pl = *L1, pr = *R1; (pl != NULL) &&
    (pr != NULL); )
switch (d1_order(L1, R1)) {
  case 1:
    pr = *(++R1); break;
  case -1:
    pl = *(++L1); break;
  case 0:
    RESET(pr, ACTIVE);
    INLINEset_or(pl, pl, pr);
    pr = *(++R1);
}
```

benchmark: gcc file name: reload.c

```
if (in != 0)
  class = PREFERRED_RELOAD_CLASS (in, class);
if (class == NO_REGS)
  .....
```

benchmark: gcc file name: cse.c

```
if (elt != 0 && elt->related_value != 0)
  relt = elt;
else if (elt == 0 && GET_CODE (x) == CONST)
  {
   rtx subexp = get_related_value (x);
   if (subexp != 0)
    relt = lookup (subexp,
        safe_hash (subexp, GET_MODE (subexp)) %
        NBUCKETS,
        GET_MODE (subexp));
  }
if (relt == 0)
 return 0;
```

benchmark: gcc file name: flow.c

```
for (j = XVECLEN (x, i) - 1; j >= 0; j--)
 {
  .....
 if (value == 0)
    value = tem;
    .....
 }
```

benchmark: gcc file name: flow.c

```
while (INSN_DELETED_P (first))
 first = NEXT_INSN (first);
while (prev != first)
 {
  prev = PREV_INSN (prev);
  PUT_CODE (prev, NOTE);
  NOTE_LINE_NUMBER (prev) = NOTE_INSN_DELETED;
  NOTE_SOURCE_FILE (prev) = 0;
 }
```

benchmark: gcc file name: cse.c

```
if (tem != 0)
    y0 = tem;
if (y0 == 0)
    return 0;
```

benchmark: gcc file name: cse.c

```
switch (i)
 {
 case 0:
  const_arg0 = const_arg;
  break;
 case 1:
  const_arg1 = const_arg;
  break;
 case 2:
  const_arg2 = const_arg;
  break;
 }
.....
switch (code)
 {
 .....
 case EQ:
  if (const_arg0 && const_arg0 == XEXP (x, 0)
  && (! (const_arg1 && const_arg1 == XEXP (x, 1))
    || (GET_CODE (const_arg0) == CONST_INT
      && GET_CODE (const_arg1) != CONST_INT)))
 .....
```

Alternative Implementations of Two-Level Adaptive Branch Prediction

Tse-Yu Yeh Patt, Y.N.
The University of Michigan;

This paper appears in: **Computer Architecture, 1992. Pr ceedings., The 19th Annual International Symposium on**

Publication Date: 1992
On page(s): 124-134

Abstract:
Not Available

Index Terms:
Not Available

Alternative Implementations of Two-Level Adaptive Branch Prediction

Tse-Yu Yeh and Yale N. Patt
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

Abstract

As the issue rate and depth of pipelining of high performance Superscalar processors increase, the importance of an excellent branch predictor becomes more vital to delivering the potential performance of a wide-issue, deep pipelined microarchitecture. We propose a new dynamic branch predictor (Two-Level Adaptive Branch Prediction) that achieves substantially higher accuracy than any other scheme reported in the literature. The mechanism uses two levels of branch history information to make predictions, the history of the last k branches encountered, and the branch behavior for the last s occurrences of the specific pattern of these k branches. We have identified three variations of the Two-Level Adaptive Branch Prediction, depending on how finely we resolve the history information gathered. We compute the hardware costs of implementing each of the three variations, and use these costs in evaluating their relative effectiveness. We measure the branch prediction accuracy of the three variations of Two-Level Adaptive Branch Prediction, along with several other popular proposed dynamic and static prediction schemes, on the SPEC benchmarks. We show that the average prediction accuracy for Two-Level Adaptive Branch Prediction is 97 percent, while the other known schemes achieve at most 94.4 percent average prediction accuracy. We measure the effectiveness of different prediction algorithms and different amounts of history and pattern information. We measure the costs of each variation to obtain the same prediction accuracy.

1 Introduction

As the issue rate and depth of pipelining of high performance Superscalar processors increase, the amount of speculative work due to branch prediction becomes much larger. Since all such work must be thrown away if the prediction is incorrect, an excellent branch predictor is vital to delivering the potential performance of a wide-issue, deep pipelined microarchitecture. Even a prediction miss rate of 5 percent results in a substantial loss in performance due to the number of instructions fetched each cycle and the number of cycles these instructions are in the pipeline before an incorrect branch prediction becomes known.

The literature is full of suggested branch prediction schemes [6, 13, 14, 17]. Some are static in that they use opcode information and profiling statistics to make predictions. Others are dynamic in that they use run-time execution history to make predictions. Static schemes can be as simple as always predicting that the branch will be taken, or can be based on the opcode, or on the direction of the branch, as in "if the branch is backward, predict taken, if forward, predict not taken" [17]. This latter scheme is effective for loop intensive code, but does not work well for programs where the branch behavior is irregular. Also, profiling [6, 13] can be used to predict branches by measuring the tendency of a branch on sample data sets and presetting a static prediction bit in the opcode according to that tendency. Unfortunately, branch behavior for the sample data may be very different from the data that appears at run-time.

Dynamic branch prediction also can be as simple as in keeping track only of the last execution of that branch instruction and predicting the branch will behave the same way, or it can be elaborate as in maintaining very large amounts of history information. In all cases, the fact that the dynamic prediction is being made on the basis of run-time history information implies that substantial additional hardware is required. J. Smith [17] proposed utilizing a branch target buffer to store, for each branch, a two-bit saturating up-down counter which collects and subsequently bases its prediction on branch history information about that branch. Lee and A. Smith proposed [14] a Static Training method which uses statistics gathered prior to execution time coupled with the history pattern of the last k run-time executions of the branch to make the next prediction as to which way that branch will go. The major disadvantage of Static Training methods has been mentioned above with respect to profiling; the pattern history statistics gathered for the sample data set may not be applicable to the data that appears at run-time.

In this paper we propose a new dynamic branch predictor that achieves substantially higher accuracy than any other scheme reported in the literature. The mechanism uses two levels of branch history information to make predictions. The first level is the history of the

© 1992 ACM 0-89791-509-7/92/0005/0124 $1.50

last k branches encountered. (Variations of our scheme reflect whether this means the actual last k branches encountered, or the last k occurrences of the same branch instruction.) The second level is the branch behavior for the last s occurrences of the specific pattern of these k branches. Prediction is based on the branch behavior for the last s occurrences of the pattern in question.

For example, suppose, for $k = 8$, the last k branches had the behavior 11100101 (where 1 represents that the branch was taken, 0 that the branch was not taken). Suppose further that $s = 6$, and that in each of the last six times the previous eight branches had the pattern 11100101, the branch alternated between taken and not taken. Then the second level would contain the history 101010. Our branch predictor would predict "taken."

The history information for level 1 and the pattern information for level 2 are collected at run time, eliminating the above mentioned disadvantages of the Static Training method. We call our method Two-Level Adaptive Branch Prediction. We have identified three variations of Two-Level Adaptive Branch Prediction, depending on how finely we resolve the history information gathered. We compute the hardware costs of implementing each of the three variations, and use these costs in evaluating their relative effectiveness.

Using trace-driven simulation of nine of the ten SPEC benchmarks [1], we measure the branch prediction accuracy of the three variations of Two-Level Adaptive Branch Prediction, along with several other popular proposed dynamic and static prediction schemes. We measure the effectiveness of different prediction algorithms and different amounts of history and pattern information. We measure the costs of each variation to obtain the same prediction accuracy. Finally we compare the Two-Level Adaptive branch predictors to the several popular schemes available in the literature. We show that the average prediction accuracy for Two-Level Adaptive Branch Prediction is about 97 percent, while the other schemes achieve at most 94.4 percent average prediction accuracy.

This paper is organized in six sections. Section two introduces our Two-Level Adaptive Branch Prediction and its three variations. Section three describes the corresponding implementations and computes the associated hardware costs. Section four discusses the Simulation model and traces used in this study. Section five reports the simulation results and our analysis. Section six contains some concluding remarks.

2 Definition of Two-Level Adaptive Branch Prediction

2.1 Overview

Two-Level Adaptive Branch Prediction uses two levels of branch history information to make predictions. The first level is the history of the last k branches encountered. (Variations of our scheme reflect whether this means the actual last k branches encountered, or the last k occurrences of the same branch instruction.) The second level is the branch behavior for the last s occurrences of the specific pattern of these k branches. Prediction is based on the branch behavior for the last s occurrences of the pattern in question.

To maintain the two levels of information, Two-Level Adaptive Branch Prediction uses two major data structures, the branch history register (HR) and the pattern history table (PHT), see Figure 1. Instead of accumulating statistics by profiling programs, the information on which branch predictions are based is collected at run-time by updating the contents of the history registers and the pattern history bits in the entries of the pattern history table depending on the outcomes of the branches. The history register is a k-bit shift register which shifts in bits representing the branch results of the most recent k branches.

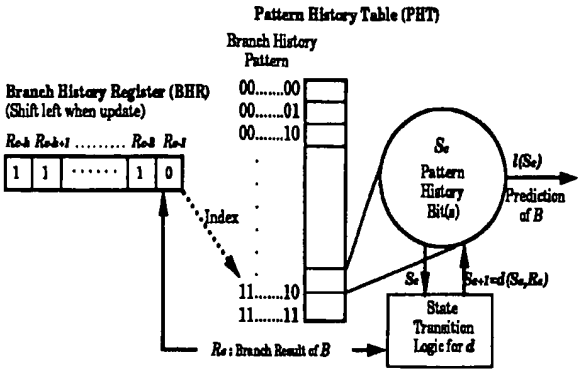


Figure 1: Structure of Two-Level Adaptive Branch Prediction.

If the branch was taken, then a "1" is recorded; if not, a "0" is recorded. Since there are k bits in the history register, at most 2^k different patterns appear in the history register. For each of these 2^k patterns, there is a corresponding entry in the pattern history table which contains branch results for the last s times the preceding k branches were represented by that specific content of the history register.

When a conditional branch B is being predicted, the content of its history register, HR, denoted as $R_{c-k}R_{c-k+1}.....R_{c-1}$, is used to address the pattern history table. The pattern history bits S_c in the addressed entry $PHT_{R_{c-k}R_{c-k+1}.....R_{c-1}}$ in the pattern history table are then used for predicting the branch. The prediction of the branch is

$$z_c = \lambda(S_c), \tag{1}$$

where λ is the prediction decision function.

After the conditional branch is resolved, the outcome R_c is shifted left into the history register HR in the least significant bit position and is also used to update the pattern history bits in the pattern history table entry $PHT_{R_{c-k}R_{c-k+1}.....R_{c-1}}$. After being

[1] The Nasa7 benchmark was not simulated because this benchmark consists of seven independent loops. It takes too long to simulate the branch behavior of these seven kernels, so we omitted these loops.

updated, the content of the history register becomes $R_{c-k+1}R_{c-k+2}.....R_c$ and the state represented by the pattern history bits becomes S_{c+1}. The transition of the pattern history bits in the pattern history table entry is done by the state transition function δ which takes in the old pattern history bits and the outcome of the branch as inputs to generate the new pattern history bits. Therefore, the new pattern history bits S_{c+1} become

$$S_{c+1} = \delta(S_c, R_c). \quad (2)$$

A straightforward combinational logic circuit is used to implement the function δ to update the pattern history bits in the entries of the pattern history table. The transition function δ, predicting function λ, pattern history bits S and the outcome R of the branch comprise a finite-state Moore machine, characterized by equations 1 and 2.

State diagrams of the finite-state Moore machines used in this study for updating the pattern history in the pattern history table entry and for predicting which path the branch will take are shown in Figure 2. The automaton *Last-Time* stores in the pattern history only the outcome of the last execution of the branch when the history pattern appeared. The next time the same history pattern appears the prediction will be what happened last time. Only one bit is needed to store that pattern history information. The automaton $A1$ records the results of the last two times the same history pattern appeared. Only when there is no taken branch recorded, the next execution of the branch when the history register has the same history pattern will be predicted as not taken; otherwise, the branch will be predicted as taken. The automaton $A2$ is a saturating up-down counter, similar to the automaton used in J. Smith's branch target buffer design for keeping branch history [17].

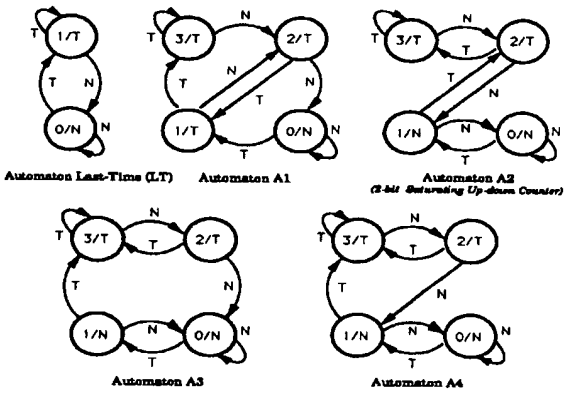


Figure 2: State diagrams of the finite-state Moore machines used for making prediction and updating the pattern history table entry.

In J. Smith's design the 2-bit saturating up-down counter keeps track of the branch history of a certain branch. The counter is incremented when the branch is taken and is decremented when the branch is not taken. The branch path of the next execution of the branch will be predicted as taken when the counter value is greater than or equal to two; otherwise, the branch will be predicted as not taken. In Two-Level Adaptive Branch Prediction, the 2-bit saturating up-down counter keeps track of the history of a certain history pattern. The counter is incremented when the result of a branch, whose history register content is the same as the pattern history table entry index, is taken; otherwise, the counter is decremented. The next time the branch has the same history register content which accesses the same pattern history table entry, the branch is predicted taken if the counter value is greater or equal to two; otherwise, the branch is predicted not taken. Automata $A3$ and $A4$ are variations of $A2$.

Both Static Training [14] and Two-Level Adaptive Branch Prediction are dynamic branch predictors, because their predictions are based on run-time information, i.e. the dynamic branch history. The major difference between these two schemes is that the pattern history information in the pattern history table changes dynamically in Two-Level Adaptive Branch Prediction but is preset in Static Training from profiling. In Static Training, the input to the prediction decision function, λ, for a given branch history pattern is known before execution. Therefore, the output of λ is determined before execution for a given branch history pattern. That is, the same branch predictions are made if the same history pattern appears at different times during execution. Two-Level Adaptive Branch Prediction, on the other hand, updates the pattern history information kept in the pattern history table with the actual results of branches. As a result, given the same branch history pattern, different pattern history information can be found in the pattern history table; therefore, there can be different inputs to the prediction decision function for Two-Level Adaptive Branch Prediction. Predictions of Two-Level Adaptive Branch Prediction change adaptively as the program executes.

Since the pattern history bits change in Two-Level Adaptive Branch Prediction, the predictor can adjust to the current branch execution behavior of the program to make proper predictions. With these run-time updates, Two-Level Adaptive Branch Prediction can be highly accurate over many different programs and data sets. Static Training, on the contrary, may not predict well if changing data sets brings about different execution behavior.

2.2 Alternative Implementations of Two-Level Adaptive Branch Prediction

There are three alternative implementations of the Two-Level Adaptive Branch Prediction, as shown in Figure 3. They are differentiated as follows:

Two-Level Adaptive Branch Prediction Using a Global History Register and a Global Pattern History Table (GAg)

In GAg, there is only a single global history register (GHR) and a single global pattern history table (GPHT) used by the Two-Level Adaptive Branch Pre-

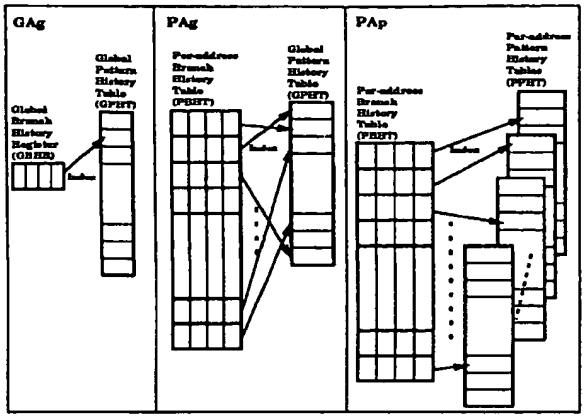


Figure 3: Global view of three variations of Two-Level Adaptive Branch Prediction.

diction. All branch predictions are based on the same global history register and global pattern history table which are updated after each branch is resolved. This variation therefore is called Global Two-Level Adaptive Branch Prediction using a global pattern history table (GAg).

Since the outcomes of different branches update the same history register and the same pattern history table, the information of both branch history and pattern history is influenced by results of different branches. The prediction for a conditional branch in this scheme is actually dependent on the outcomes of other branches.

Two-Level Adaptive Branch Prediction Using a Per-address Branch History Table and a Global Pattern History Table (PAg)

In order the reduce the interference in the first level branch history information, one history register is associated with each distinct static conditional branch to collect branch history information individually. The history registers are contained in a per-address branch history table (PBHT) in which each entry is accessible by one specific static branch instruction and is accessed by branch instruction addresses. Since the branch history is kept for each distinct static conditional branch individually and all history registers access the same global pattern history table, this variation is called Per-address Two-Level Adaptive Branch Prediction using a global pattern history table (PAg).

The execution results of a static conditional branch update the branch's own history register and the global pattern history table. The prediction for a conditional branch is based on the branch's own history and the pattern history bits in the global pattern history table entry indexed by the content of the branch's history register. Since all branches update the same pattern history table, the pattern history interference still exists.

Two-Level Adaptive Branch Prediction Using Per-address Branch History Table and Per-address Pattern History Tables (PAp)

In order to completely remove the interference in both levels, each static branch has its own pattern history table a set of which is called a per-address pattern history table (PPHT). Therefore, a per-address history register and a per-address pattern history table are associated with each static conditional branch. All history registers are grouped in a per-address branch history table. Since this variation of Two-Level Adaptive Branch Prediction keeps separate history and pattern information for each distinct static conditional branch, it is called Per-address Two-Level Adaptive Branch Prediction using Per-address pattern history tables (PAp).

3 Implementation Considerations

3.1 Pipeline Timing of Branch Prediction and Information Update

Two-Level Adaptive Branch Prediction requires two sequential table accesses to make a prediction. It is difficult to squeeze the two accesses into one cycle. High performance requires that prediction be made within one cycle from the time the branch address is known. To satisfy this requirement, the two sequential accesses are performed in two different cycles as follows: When a branch result becomes known, the branch's history register is updated. In the same cycle, the pattern history table can be accessed for the next prediction with the updated history register contents derived by appending the result to the old history. The prediction fetched from the pattern history table is then stored along with the branch's history in the branch history table. The pattern history can also be updated at that time. The next time that branch is encountered, the prediction is available as soon as the branch history table is accessed. Therefore, only one cycle latency is incurred from the time the branch address is known to the time the prediction is available.

Sometimes the previous branch results may not be ready before the prediction of a subsequent branch takes place. If the obsolete branch history is used for making the prediction, the accuracy is degraded. In such a case, the predictions of the previous branches can be used to update the branch history. Since the prediction accuracy of Two-Level Adaptive Branch Prediction is very high, prediction is enhanced by updating the branch history speculatively. The update timing for the pattern history table, on the other hand, is not as critical as that of the branch history; therefore, its update can be delayed until the branch result is known. With speculative updating, when a misprediction occurs, the branch history can either be reinitialized or repaired depending on the hardware budget available to the branch predictor. Also, if two instances of the same static branch occur in consecutive cycles, the latency of prediction can be reduced for the second branch by using the prediction fetched from the pattern history table directly.

3.2 Target Address Caching

After the direction of a branch is predicted, there is still the possibility of a pipeline bubble due to the time it takes to generate the target address. To eliminate

this bubble, we cache the target addresses of branches. One extra field is required in each entry of the branch history table for doing this. When a branch is predicted taken, the target address is used to fetch the following instructions; otherwise, the fall-through address is used.

Caching the target addresses makes prediction in consecutive cycles possible without any delay. This also requires the branch history table to be accessed by the fetching address of the instruction block rather than by the address of the branch in the instruction block being fetched because the branch address is not known until the instruction block is decoded. If the address hits in the branch history table, the prediction of the branch in the instruction block can be made before the instructions are decoded. If the address misses in the branch history table, either there is no branch in the instruction block fetched in that cycle or the branch history information is not present in the branch history table. In this case, the next sequential address is used to fetch new instructions. After the instructions are decoded, if there is a branch in the instruction block and if the instruction block address missed in the branch history table, static branch prediction is used to determine whether or not the new instructions fetched from the next sequential address should be squashed.

3.3 Per-address Branch History Table Implementation

PAg and PAp branch predictors all use per-address branch history tables in their structure. It is not feasible to have a branch history table large enough to hold all branches' execution history in real implementations. Therefore, a practical approach for the per-address branch history table is proposed here.

The per-address branch history table can be implemented as a set-associative or direct-mapped cache. A fixed number of entries in the table are grouped together as a set. Within a set, a Least-Recently-Used (LRU) algorithm is used for replacement. The lower part of a branch address is used to index into the table and the higher part is stored as a tag in the entry associated with that branch. When a conditional branch is to be predicted, the branch's entry in the branch history table is located first. If the tag in the entry matches the accessing address, the branch information in the entry is used to predict the branch. If the tag does not match the address, a new entry is allocated for the branch.

In this study, both the above practical approach and an Ideal Branch History Table (IBHT), in which there is a history register for each static conditional branch, were simulated for Two-Level Adaptive Branch Prediction. The branch history table was simulated with four configurations: 4-way set-associative 512-entry, 4-way set-associative 256-entry, direct-mapped 512-entry and direct-mapped 256-entry caches. The IBHT simulation data is provided to show the accuracy loss due to the history interference in a practical branch history table implementations.

3.4 Hardware Cost Estimates

The chip area required for a run-time branch prediction mechanism is not inconsequential. The following hardware cost estimates are proposed to characterize the relative costs of the three variations. The branch history table and the pattern history table are the two major parts. Detailed items include storage space for keeping history information, prediction bits, tags, and LRU bits and the accessing and updating logic of the tables. The accessing and updating logic consists of comparators, MUXes, LRU bits incrementors, and address decoders for the branch history table, and address decoders and pattern history bit update circuits for the pattern history table. The storage space for caching target addresses is not included in the following equations because it is not required for the branch predictor.

Assumptions of these estimates are:

- There are a address bits, a subset of which is used to index the branch history table and the rest are stored as a tag in the indexed branch history table entry.
- In an entry of the branch history table, there are fields for branch history, an address tag, a prediction bit, and LRU bits.
- The branch history table size is h.
- The branch history table is 2^j-way set-associative.
- Each history register contains k bits.
- Each pattern history table entry contains s bits.
- Pattern history table set size is p. (In PAp, p is equal to the size of the branch history table, h, while in GAg and PAg, p is always equal to one.)
- C_s, C_d, C_c, C_m, C_{sh}, C_i, and C_a are the constant base costs for the storage, the decoder, the comparator, the multiplexer, the shifter, the incrementor, and the finite-state machine.

Furthermore, i is equal to $log_2 h$ and is a non-negative integer. When there are k bits in a history register, a pattern history table always has 2^k entries.

The hardware cost of Two-Level Adaptive Branch Prediction is as follows:

$$
\begin{aligned}
&Cost_{Scheme}(BHT(h,j,k), p \times PHT(2^k, s)) \\
&= Cost_{BHT}(h,j,k) + p \times Cost_{PHT}(2^k, s) \\
&= \{BHT_{Storage\_Space} + BHT_{Accessing\_Logic} + \\
&\quad BHT_{Updating\_Logic}\} + p \times \{PHT_{Storage\_Space} + \\
&\quad PHT_{Accessing\_Logic} + PHT_{Updating\_Logic}\} \\
&= \{[h \times (Tag_{(a-i+j)\_bit} + HR_{k\_bit} + Prediction\_Bit_{1\_bit} \\
&\quad + LRU\_Bits_{j\_bit})] + \\
&\quad [1 \times Address\_Decoder_{i\_bit} + 2^j \times \\
&\quad Comparators_{(a-i+j)\_bit} + 1 \times 2^j X1\_MUX_{k\_bit}] + \\
&\quad [h \times Shifter_{k\_bit} + 2^j \times LRU\_Incrementors_{j\_bit}]\} + \\
&\quad p \times \{[2^k \times History\_Bits_{s\_bit}] + \\
&\quad [1 \times Address\_Decoder_{k\_bit}] + [State\_Updater_{s\_bit}]\}
\end{aligned}
$$

$$= \{h \times [(a-i+j)+k+1+j] \times C_s + [h \times C_d + 2^j \times (a-i+j) \times C_c + 2^j \times k \times C_m] + [h \times k \times C_{sh} + 2^j \times j \times C_i]\} + p \times \{[2^k \times s \times C_s] + [2^k \times C_d] + [s \times 2^{s+1} \times C_a]\}, \quad a+j \geq i. \tag{3}$$

In GAg, only one history register and one global pattern history table are used, so h and p are both equal to one. No tag and no branch history table accessing logic are necessary for the single history register. Besides, pattern history state updating logic is small compared to the other two terms in the pattern history table cost. Therefore, cost estimation function for GAg can be simplified from Function 3 to the following Function:

$$\begin{aligned} & Cost_{GAg}(BHT(1, ,k), 1 \times PHT(2^k, s)) \\ &= Cost_{BHT}(1, ,k) + 1 \times Cost_{PHT}(2^k, s) \\ &\simeq \{[k+1] \times C_s + k \times C_{sh}\} + \{2^k \times (s \times C_s + C_d)\} \end{aligned} \tag{4}$$

It is clear to see that the cost of GAg grows exponentially with respect to the history register length.

In PAg, only one pattern history table is used, so p is equal to one. Since j and s are usually small compared to the other variables, by using Function 3, the estimated cost for PAg using a branch history table is as follows:

$$\begin{aligned} & Cost_{PAg}(BHT(h,j,k), 1 \times PHT(2^k, s)) \\ &= Cost_{BHT}(h,j,k) + 1 \times Cost_{PHT}(2^k, s) \\ &\simeq \{h \times [(a+2 \times j+k+1-i) \times C_s + C_d + k \times C_{sh}]\} + \{2^k \times (s \times C_s + C_d)\}, \quad a+j \geq i. \end{aligned} \tag{5}$$

The cost of a PAg scheme grows exponentially with respect to the history register length and linearly with respect to the branch history table size.

In a PAp scheme using a branch history table as defined above, h pattern history tables are used, so p is equal to h. By using Function 3, the estimated cost for PAp is as follows:

$$\begin{aligned} & Cost_{PAp}(BHT(h,j,k), h \times PHT(2^k, s)) \\ &= Cost_{BHT}(h,j,k) + h \times Cost_{PHT}(2^k, s) \\ &\simeq \{h \times [(a+2 \times j+k+1-i) \times C_s + C_d + k \times C_{sh}]\} + h \times \{2^k \times (s \times C_s + C_d)\}, \quad a+j \geq i. \end{aligned} \tag{6}$$

When the history register is sufficiently large, the cost of a PAp scheme grows exponentially with respect to the history register length and linearly with respect to the branch history table size. However, the branch history table size becomes a more dominant factor than it is in a PAg scheme.

4 Simulation Model

Trace-driven simulations were used in this study. A Motorola 88100 instruction level simulator is used for generating instruction traces. The instruction and address traces are fed into the branch prediction simulator which decodes instructions, predicts branches, and verifies the predictions with the branch results to collect statistics for branch prediction accuracy.

4.1 Description of Traces

Nine benchmarks from the SPEC benchmark suite are used in this branch prediction study. Five are floating point benchmarks and four are integer benchmarks. The floating point benchmarks include *doduc*, *fpppp*, *matrix*300, *spice2g6* and *tomcatv* and the integer ones include *eqntott*, *espresso*, *gcc*, and *li*. *Nasa7* is not included because it takes too long to capture the branch behavior of all seven kernels.

Among the five floating point benchmarks, *fpppp*, *matrix*300 and *tomcatv* have repetitive loop execution; thus, a very high prediction accuracy is attainable, independent of the predictors used. *Doduc*, *spice2g6* and the integer benchmarks are more interesting. They have many conditional branches and irregular branch behavior. Therefore, it is on the integer benchmarks where a branch predictor's mettle is tested.

Since this study of branch prediction focuses on the prediction for conditional branches, all benchmarks were simulated for twenty million conditional branch instructions except *gcc* which finished before twenty million conditional branch instructions are executed. *Fpppp*, *matrix*300, and *tomcatv* were simulated for 100 million instruction because of their regular branch behavior through out the programs. The number of static conditional branches in the instruction traces of the benchmarks are listed in Table 1. History register hit rate usually depends on the number of static branches in the benchmarks. The testing and training data sets for each benchmark used in this study are listed in Table 2.

Benchmark Name	Number of Static Cnd. Br.	Benchmark Name	Number of Static Cnd. Br.
eqntott	277	espresso	556
gcc	6922	li	489
doduc	1149	fpppp	653
matrix300	213	spice2g6	606
tomcatv	370		

Table 1: Number of static conditional branches in each benchmark.

Benchmark Name	Training Data Set	Testing Data Set
eqntott	NA	int_pri_3.eqn
espresso	cps	bca
gcc	cexp.i	dbxout.i
xlisp	tower of hanoi	eight queens
doduc	tiny doducin	doducin
fpppp	NA	natoms
matrix300	NA	Built-in
spice2g6	short greycode.in	greycode.in
tomcatv	NA	Built-in

Table 2: Training and testing data sets of benchmarks.

In the traces generated with the testing data sets, about 24 percent of the dynamic instructions for the integer benchmarks and about 5 percent of the dynamic instructions for the floating point benchmarks are branch instructions. Figure 4 shows about 80 percent of the dynamic branch instructions are conditional branches; therefore, the prediction mechanism for conditional branches is the most important among the prediction mechanisms for different classes of branches.

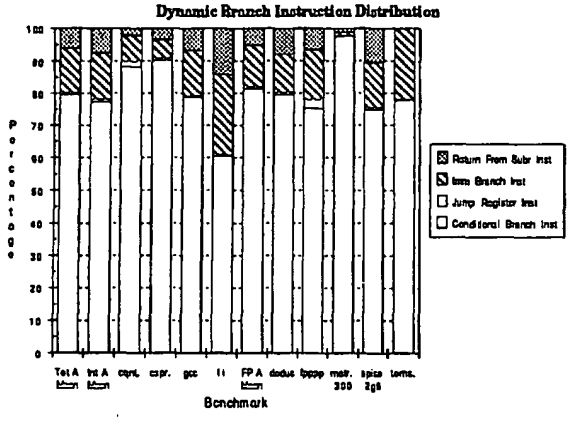


Figure 4: Distribution of dynamic branch instructions.

4.2 Characterization of Branch Predictors

The three variations of Two-Level Adaptive Branch Prediction were simulated with several configurations. Other known dynamic and static branch predictors were also simulated. The configurations of the dynamic branch predictors are shown in Table 3. In order to distinguish the different schemes we analyzed, the following naming convention is used: *Scheme*(*History*(*Size*, *Associativity*, *Entry_Content*), *Pattern_Table_Set_Size* × *Pattern*(*Size*, *Entry_Content*), *Context_Switch*). If a predictor does not have a certain feature in the naming convention, the corresponding field is left blank.

Scheme specifies the scheme, for example, GAg, PAg, PAp or Branch Target Buffer design (BTB) [17]. In *History*(*Size*, *Associativity*, *Entry_Content*), *History* is the entity used to keep history information of branches, for example, HR (A single history register), IBHT, or BHT. *Size* specifies the number of entries in that entity, *Associativity* is the associativity of the table, and *Entry_Content* specifies the content in each branch history table entry. When *Associativity* is set to 1, the branch history table is direct-mapped. The content of an entry in the branch history table can be any automaton shown in Figure 2 or simply a history register.

In *Pattern_Table_Set_Size* × *Pattern*(*Size*, *Entry_Content*), *Pattern_Table_Set_Size* is the number of pattern history tables used in the scheme, *Pattern* is the implementation for keeping pattern history information, *Size* specifies the number of entries in the implementation, and *Entry_Content* specifies the content in each entry. The content of an entry in the pattern history table can be any automaton shown in Figure 2. For Branch Target Buffer designs, the *Pattern* part is not included, because there is no pattern history information kept in their designs. *Context_Switch* is a flag for context switches. When *Context_Switch* is specified as c, context switches are simulated. If it is not specified, no context switches are simulated.

Since there are more taken branches than not taken branches according to our simulation results, a history register in the branch history table is initialized to all 1's when a miss on the branch history table occurs. After the result of the branch which causes the branch history table miss is known, the result bit is extended throughout the history register. A context switch results in flushing and reinitialization of the branch history table.

Model Name	BHT Config. # of Entr.	Asc	Entry Cont.	PHT Set Size	PHT Config. # of Entr.	Entry Cont.
GAg(HR(1, ,r-sr), 1xPHT(2^r,A2),[c])	1		r-bit sr	1	2^r	Atm A2
PAg(BHT(256,1,r-sr), 1xPHT(2^r,A2),[c])	256	1	r-bit sr	1	2^r	Atm A2
PAg(BHT(256,4,r-sr), 1xPHT(2^r,A2),[c])	256	4	r-bit sr	1	2^r	Atm A2
PAg(BHT(512,1,r-sr), 1xPHT(2^r,A2),[c])	512	1	r-bit sr	1	2^r	Atm A2
PAg(BHT(512,4,r-sr), 1xPHT(2^r,A1),[c])	512	4	r-bit sr	1	2^r	Atm A1
PAg(BHT(512,4,r-sr), 1xPHT(2^r,A2),[c])	512	4	r-bit sr	1	2^r	Atm A2
PAg(BHT(512,4,r-sr), 1xPHT(2^r,A3),[c])	512	4	r-bit sr	1	2^r	Atm A3
PAg(BHT(512,4,r-sr), 1xPHT(2^r,A4),[c])	512	4	r-bit sr	1	2^r	Atm A4
PAg(BHT(512,4,r-sr), 1xPHT(2^r,LT),[c])	512	4	r-bit sr	1	2^r	Atm LT
PAg(IBHT(inf, ,r-sr), 1xPHT(2^r,A2),[c])	∞		r-bit sr	1	2^r	Atm A2
PAp(BHT(512,4,r-sr), 512xPHT(2^r,A2),[c])	512	4	r-bit sr	512	2^r	Atm A2
GSg(HR(1, ,r-sr), 1xPHT(2^r,PB),[c])	1		r-bit sr	1	2^r	PB
PSg(BHT(512,4,r-sr), 1xPHT(2^r,PB),[c])	512	4	r-bit sr	1	2^r	PB
BTB(BHT(512,4,A2), ,[c])	512	4	Atm A2			
BTB(BHT(512,4,LT), ,[c])	512	4	Atm LT			

Asc - *Table Set-Associativity*, **Atm** - *Automaton*, **BHT** - *Branch History Table*, **BTB** - *Branch Target Buffer Design*, **Config.** - *Configuration*, **Entr.** - *Entries*, **GAg** - *Global Two-Level Adaptive Branch Prediction Using a Global Pattern History Table*, **GSg** - *Global Static Training Using a Preset Global Pattern History Table*, **IBHT** - *Ideal Branch History Table*, **inf** - *Infinite*, **LT** - *Last-Time*, **PAg** - *Per-address Two-Level Adaptive Branch Prediction Using a Global Pattern History Table*, **PAp** - *Per-address Two-Level Adaptive Branch Prediction Using Per-address Pattern History Tables*, **PB** - *Preset Prediction Bit*, **PSg** - *Per-address Static Training Using a Preset Global Pattern History Table*, **PHT** - *Pattern History Table*, **sr** - *Shift Register*.

Table 3: Configurations of simulated branch predictors.

The pattern history bits in the pattern history table entries are also initialized at the beginning of execution. Since taken branches are more likely for those pattern history tables using automata *A1*, *A2*, *A3*, and *A4*, all entries are initialized to state 3. For *Last-Time*, all entries are initialized to state 1 such that the branches at

the beginning of execution will be more likely to be predicted taken. It is not necessary to reinitialize pattern history tables during execution.

In addition to the Two-Level Adaptive schemes, Lee and A. Smith's Static Training schemes, Branch Target Buffer designs, and some dynamic and static branch prediction schemes were simulated for comparison purposes. Lee and A. Smith's Static Training scheme is similar in structure to the Per-address Two-Level Adaptive scheme with an IBHT but with the important difference that the prediction for a given pattern is pre-determined by profiling. In this study, Lee and A. Smith's Static Training is identified as PSg, meaning per-address Static Training using a global preset pattern history table. Similarly, the scheme which has a similar structure to GAg but with the difference that the second-level pattern history information is collected from profiling is abbreviated PSg, meaning Global Static Training using a preset global pattern history table. Per-address Static Training using per-address pattern history tables (PSp) is another application of Static Training to a different structure; however, this scheme requires a lot of storage to keep track of pattern behavior of all branches statically. Therefore, no PSp schemes were simulated in this study. Lee and A. Smith's Static Training schemes were simulated with the same branch history table configurations as used by the Two-Level Adaptive schemes for a fair comparison. The cost to implement Static Training is not less expensive than the cost to implement the Two-Level Adaptive Scheme because the branch history table and the pattern history table required by both schemes are similar. In Static Training, before program execution starts, extra time is needed to load the preset pattern prediction bits into the pattern history table.

Branch Target Buffer designs were simulated with automata $A2$ and *Last-Time*. The static branch prediction schemes simulated include the Always Taken, Backward Taken and Forward Not Taken, and a profiling scheme. Always Taken scheme predicts taken for all branches. Backward Taken and Forward Not Taken (BTFN) scheme predicts taken if a branch branches backward and not taken if the branch branches forward. The BTFN scheme is effective for loop-bound programs, because it mispredicts only once in the execution of a loop. The profiling scheme counts the frequency of taken and not-taken for each static branch in the profiling execution. The predicted direction of a branch is the one the branch takes most frequently. The profiling information of a program executed with a training data set is used for branch predictions for the program executed with testing data sets, thus calculating the prediction accuracy.

5 Branch Prediction Simulation Results

Figures 5 through 11 show the prediction accuracy of the branch predictors described in the previous session on the nine SPEC benchmarks. "Tot GMean" is the geometric mean across all the benchmarks, "Int GMean" is the geometric mean across all the integer benchmarks, and "FP GMean" is the geometric mean across all the floating point benchmarks. The vertical axis shows the prediction accuracy scaled from 76 percent to 100 percent.

5.1 Evaluation of the Parameters of the Two-Level Adaptive Branch Prediction Branch Prediction

The three variations of Two-Level Adaptive Branch Prediction were simulated with different history register lengths to assess the effectiveness of increasing the recorded history length. The PAg and PAp schemes were each simulated with an ideal branch history table (IBHT) and with practical branch history tables to show the effect of the branch history table hit ratio.

5.1.1 Effect of Pattern History Table Automaton

Figure 5 shows the efficiency of using different finite-state automata. Five automata $A1$, $A2$, $A3$, $A4$, and *Last-Time* were simulated with a PAg branch predictor, having 12-bit history registers in a four-way set-associative 512-entry BHT. $A1$, $A2$, $A3$, and $A4$ all perform better than *Last-Time*. The four-state automata $A1$, $A2$, $A3$, and $A4$ maintain more history information than *Last-Time* which only records what happened the last time; they are therefore more tolerant to the deviations in the execution history. Among the four-state automata, $A1$ performs worse than the others. The performance of $A2$, $A3$, and $A4$ are very close to each other; however, $A2$ usually performs best. In order to show the following figures clearly, each Two-Level Adaptive Scheme is shown with automaton $A2$.

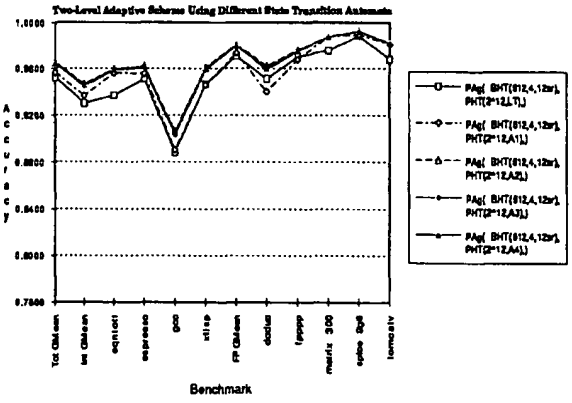


Figure 5: Comparison of Two-Level Adaptive Branch Predictors using different finite-state automata.

5.1.2 Effect of History Register Length

Three variations using history registers of the same length

Figure 6 shows the effects of history register length on the prediction accuracy of Two-Level Adaptive schemes. Every scheme in the graph was simulated with the same history register length. Among the variations, PAp performs the best, PAg the second, and GAg the worst.

GAg is not effective with 6-bit history registers, because every branch updates the same history register, causing excessive interference. PAg performs better than GAg, because it has a branch history table which reduces the interference in branch history. PAp predicts the best, because the interference in the pattern history is removed.

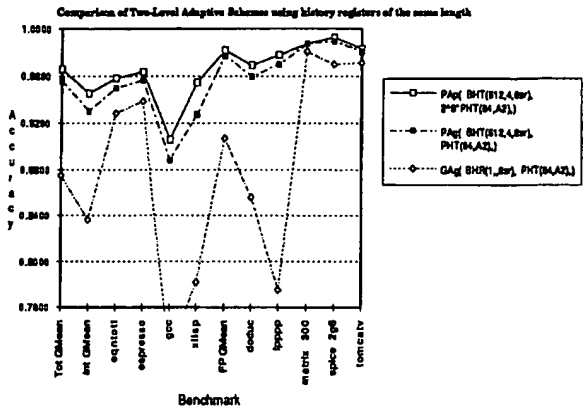


Figure 6: Comparison of the Two-Level Adaptive schemes using history registers of the same length.

Effects of various history register lengths

To further investigate the effect of history register length, Figure 7 shows the accuracy of GAg with various history register lengths. There is an increase of 9 percent in accuracy by lengthening the history register from 6 bits to 18 bits. The effect of history register length is obvious on GAg schemes. The history register length has smaller effect on PAg schemes and even smaller effect on PAp schemes because of the less interference in the branch history and pattern history and their effectiveness with short history registers.

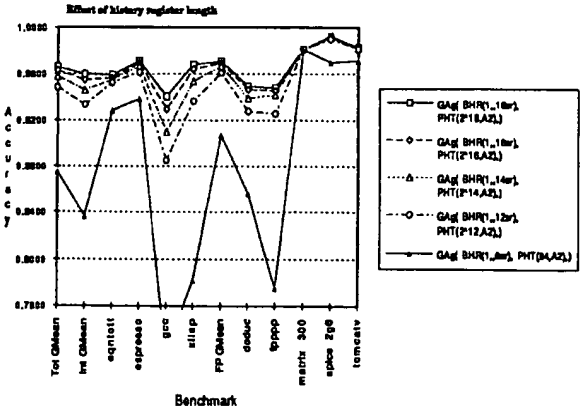


Figure 7: Effect of various history register lengths on GAg schemes.

5.1.3 Hardware Cost Efficiency of Three Variations

In Figure 6, prediction accuracy for the schemes with the same history register length were compared. However, the various Two-Level Adaptive schemes have different costs. PAp is the most expensive, PAg the second, and GAg the least, as you would expect. When evaluating the three variations of Two-Level Adaptive Branch Prediction, it is useful to know which variation is the least expensive when they predict with approximately the same accuracy.

Figure 8 illustrates three schemes which achieve about 97 percent prediction accuracy. One scheme is chosen for each variation to show the variation's configuration requirements to obtain that prediction accuracy. To achieve 97 percent prediction accuracy, GAg requires an 18-bit history register, PAg requires 12-bit history registers, and PAp requires 6-bit history registers. According to our cost estimates, PAg is the cheapest among these three. GAg's pattern history table is expensive when a long history register is used. PAp is expensive due to the required multiple pattern history tables.

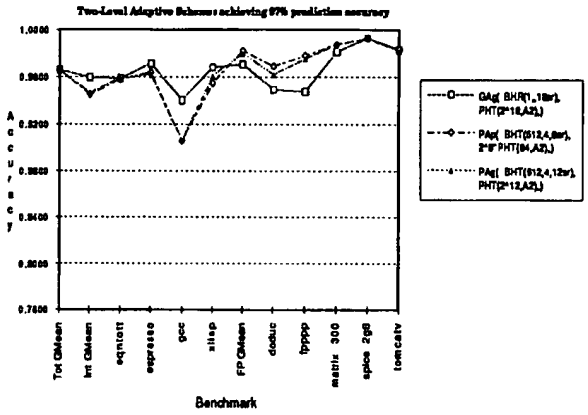


Figure 8: The Two-Level Adaptive schemes achieve about 97 percent prediction accuracy.

5.1.4 Effect of Context Switch

Since Two-Level Adaptive Branch Prediction uses the branch history table to keep track of branch history, the table needs to be flushed during a context switch. Figure 9 shows the difference in the prediction accuracy for three schemes simulated with and without context switches. During the simulation, whenever a trap occurs in the instruction trace or every 500,000 instructions if no trap occurs, a context switch is simulated. After a context switch, the pattern history table is not re-initialized, because the pattern history table of the saved process is more likely to be similar to the current process's pattern history table than to a re-initialized pattern history table. The value 500,000 is derived by assuming that a 50 MHz clock is used and context switches occur every 10 ms in a 1 IPC machine. The average accuracy degradations for the three schemes are

all less than 1 percent. The accuracy degradations for *gcc* when PAg and PAp are used are much greater than those of the other programs because of the large number of traps in *gcc*. However, the excessive number of traps do not degrade the prediction accuracy of the GAg scheme, because an initialized global history register can be refilled quickly. The prediction accuracy of *fpppp* using GAg actually increases when context switches are simulated. There are very few conditional branches in *fpppp* and all the conditional branches have regular behavior; therefore, initializing the global history register helps clear out the noise.

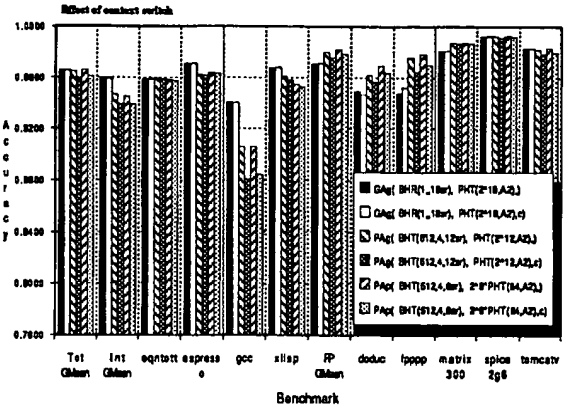


Figure 9: Effect of context switch on prediction accuracy.

5.1.5 Effect of Branch History Table Implementation

Figure 10 illustrates the effects of the size and associativity of the branch history table in the presence of context switches. Four practical branch history table implementations and an ideal branch history table were simulated. The four-way set-associative 512-entry branch history table's performance is very close to that of the ideal branch history table, because most branches in the programs can fit in the table. Prediction accuracy decreases as table miss rate increases, which is also seen in the PAp schemes.

5.2 Comparison of Two-Level Adaptive Branch Prediction and Other Prediction schemes

Figure 11 compares the branch prediction schemes. The PAg scheme which achieves 97 percent prediction accuracy is chosen for comparison with other well-known schemes, because it costs the least among the three variations of Two-Level Adaptive Branch Prediction.

The 4-way set-associative 512-entry BHT is selected to be used by all schemes which keep the first-level branch history information, because it is simple enough to be implemented. The Two-Level Adaptive scheme and the Static Training scheme were chosen on the basis of similar costs.

The top curve is achieved by the Two-Level Adaptive scheme whose prediction accuracy is about 97 percent.

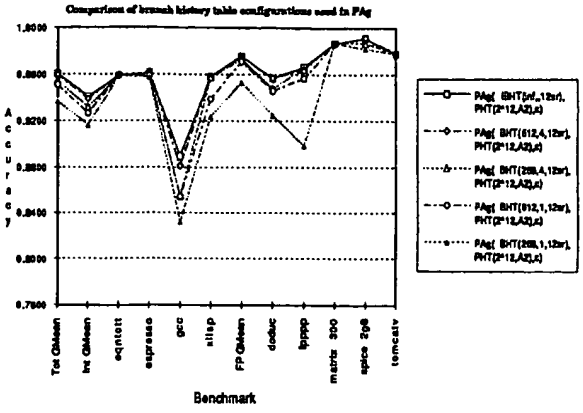


Figure 10: Effect of branch history table implementation on PAg schemes.

Since the data for the Static Training schemes are not complete due to the unavailability of appropriate data sets, the data points for *eqntott*, *fpppp*, *matrix*300 , and *tomcatv* are not graphed. PSg is about 1 to 4 percent lower than the top curve for the benchmarks that are available and GSg is about 4 to 19 percent lower with average prediction accuracy of 94.4 percent and 89 percent individually. Note that their accuracy depends greatly on the similarities between the data sets used for training and testing. The prediction accuracy for the branch target buffer using 2-bit saturating up-down counters [17] is around 93 percent. The Profiling scheme achieves about 91 percent prediction accuracy. The branch target buffer using *Last-Time* achieves about 89 percent prediction accuracy. Most of the prediction accuracy curves of BTFN and Always Taken are below the base line (76 percent). BTFN's average prediction accuracy is about 68.5 percent and Always Taken's is about 62.5 percent. In this figure, the Two-Level Adaptive scheme is superior to the other schemes by at least 2.6 percent.

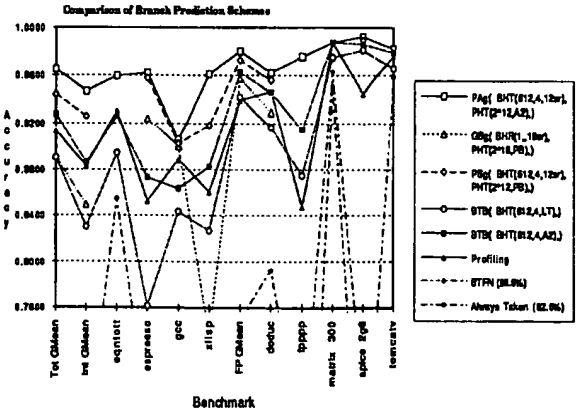


Figure 11: Comparison of branch prediction schemes.

6 Concluding Remarks

In this paper we have proposed a new dynamic branch predictor (Two-Level Adaptive Branch Prediction) that achieves substantially higher accuracy than any other scheme that we are aware of. We computed the hardware costs of implementing three variations of this scheme and determined that the most effective implementation of Two-Level Adaptive Branch Prediction utilizes a per-address branch history table and a global pattern history table.

We have measured the prediction accuracy of the three variations of Two-Level Adaptive Branch Prediction and several other popular proposed dynamic and static prediction schemes using trace-driven simulation of nine of the ten SPEC benchmarks. We have shown that the average prediction accuracy for Two-Level Adaptive Branch Prediction is about 97 percent, while the other known schemes achieve at most 94.4 percent average prediction accuracy.

We have measured the effects of varying the parameters of the Two-Level Adaptive predictors. We noted the sensitivity to k, the length of the history register, and s, the size of each entry in the pattern history table. We reported on the effectiveness of the various prediction algorithms that use the pattern history table information. We showed the effects of context switching.

Finally, we should point out that we feel our 97 percent prediction accuracy figures are not good enough and that future research in branch prediction is still needed. High performance computing engines in the future will increase the issue rate and the depth of the pipeline, which will combine to increase further the amount of speculative work that will have to be thrown out due to a branch prediction miss. Thus, the 3 percent prediction miss rate needs improvement. We are examining that 3 percent to try to characterize it and hopefully reduce it.

Acknowledgments The authors wish to acknowledge with gratitude the other members of the HPS research group at Michigan for the stimulating environment they provide, and in particular, for their comments and suggestions on this work. We are also grateful to Motorola Corporation for technical and financial support, and to NCR Corporation for the gift of an NCR Tower, Model No. 32, which was very useful in our work.

References

[1] T-Y Yeh and Y.N. Patt, "Two-Level Adaptive Branch Prediction", *Technical Report CSE-TR-117-91, Computer Science and Engineering Division, Department of EECS, The University of Michigan*, (Nov. 1991).

[2] T-Y Yeh and Y.N. Patt, "Two-Level Adaptive Branch Prediction", *The 24th ACM/IEEE International Symposium and Workshop on Microarchitecture* , (Nov. 1991), pp. 51-61.

[3] M. Butler, T-Y Yeh, Y.N. Patt, M. Alsup, H. Scales, and M. Shebanow, "Instruction Level Parallelism is Greater Than Two", *Proceedings of the 18th International Symposium on Computer Architecture*, (May. 1991), pp. 276-286.

[4] D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns" , *Proceedings of the 18th International Symposium on Computer Architecture*, (May 1991), pp. 34-42.

[5] Motorola Inc., "M88100 User's Manual", *Phoenix, Arizona*, (March 13, 1989).

[6] W.W. Hwu, T.M.Conte, and P.P.Chang, "Comparing Software and Hardware Schemes for Reducing the Cost of Branches", *Proceedings of the 16th International Symposium on Computer Architecture*, (May 1989).

[7] N.P. Jouppi and D. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (April 1989), pp. 272-282.

[8] D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processors ", *IEEE Computer*, (July 1988), pp.47-55.

[9] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-order Execution Machines", *IEEE Transactions on Computers*, (December 1987), pp.1496-1514.

[10] P. G. Emma and E. S. Davidson, "Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance" , *IEEE Transactions on Computers*, (July 1987), pp.859-876.

[11] J. A. DeRosa and H. M. Levy, "An Evaluation of Branch Architectures ", *Proceedings of the 14th International Symposium on Computer Architecture*, (June 1987), pp.10-16.

[12] D.R. Ditzel and H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", *Proceedings of the 14th International Symposium on Computer Architecture*, (June 1987), pp.2-9.

[13] S. McFarling and J. Hennessy, "Reducing the Cost of Branches", *Proceedings of the 13th International Symposium on Computer Architecture*, (1986), pp.396-403.

[14] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *IEEE Computer*, (January 1984), pp.6-22.

[15] T.R. Gross and J. Hennessy, "Optimizing Delayed Branches", *Proceedings of the 15th Annual Workshop on Microprogramming*, (Oct. 1982), pp.114-120.

[16] D.A. Patterson and C.H. Sequin, "RISC-I: A Reduced Instruction Set VLSI Computer", *Proceedings of the 8th International Symposium on Computer Architecture*, (May. 1981), pp.443-458.

[17] J.E. Smith, "A Study of Branch Prediction Strategies", *Proceedings of the 8th International Symposium on Computer Architecture*, (May. 1981), pp.135-148.

[18] T. C. Chen, "Parallelism, Pipelining and Computer Efficiency", *Computer Design*, Vol. 10, No. 1, (Jan. 1971), pp.69-74.

A comparative analysis of schemes for correlated branch prediction

Full text Pdf (1.50 MB)

Source **International Conference on Computer Architecture** archive

Proceedings of the 22nd annual international symposium on Computer architec

S. Margherita Ligure, Italy
Pages: 276 - 286
Year of Publication: 1995
ISBN:0-89791-698-0
Also published in ...

Authors Cliff Young
Nicolas Gloy
Michael D. Smith

Sponsors IEEE-CS\TCCA : TC on Computer Arhitecture
SIGARCH: ACM Special Interest Group on Computer Architecture

Publisher ACM Press New York, NY, USA

ABSTRACT

Modern high-performance architectures require extremely accurate branch predictio
performance limitations of conditional branches. We present a framework that categ
schemes by the way in which they partition dynamic branches and by the kind of pre
The framework allows us to compare and contrast branch prediction schemes, and t

work. We use the framework to show how a static correlated branch prediction sche bias and thus improves overall branch prediction accuracy. We also use the framewc fundamental differences between static and dynamic correlated branch prediction sc shows that there is room to improve the prediction accuracy of existing branch predi

↑ REFERENCES

Note: OCR errors may be found in this Reference List extracted from the full text art expose the complete List rather than only correct and linked references.

1 V. Bala, personal communication, Oct. 1994.

2 Thomas Ball , James R. Larus, Branch prediction for free, Proceedings of the conf language design and implementation, p.300-313, June 21-25, 1993, Albuquerque, N States

3 Jon Bentley, Programming pearls, ACM Press, New York, NY, 1986

4 Po-Yung Chang , Eric Hao , Tse-Yu Yeh , Yale Patt, Branch classification: a new m branch predictor performance, Proceedings of the 27th annual international symposi Microarchitecture, p.22-31, November 30-December 02, 1994, San Jose, California,

5 Joseph A. Fisher , Stefan M. Freudenberger, Predicting conditional branch directic of a program, Proceedings of the fifth international conference on Architectural supp languages and operating systems, p.85-95, October 12-15, 1992, Boston, Massachu

6 P. P. Chang , W.-W. Hwu, Inline function expansion for compiling C programs, Pr SIGPLAN '89 Conference on Programming language design and implementation, p.2 1989, Portland, Oregon, United States

7 J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Desi Jan. 1984.

8 Scott A. Mahlke , Richard E. Hank , Roger A. Bringmann , John C. Gyllenhaal , Da Wen-mei W. Hwu, Characterizing the impact of predicated execution on branch pred the 27th annual international symposium on Microarchitecture, p.217-227, Novembe 1994, San Jose, California, United States

9 S. McFarling, "Combining Branch Predictors," WRL Technical Note TN-36, June 19

10 S. McFarling , J. Hennesey, Reducing the cost of branches, Proceedings of the 1 symposium on Computer architecture, p.396-403, June 02-05, 1986, Tokyo, Japan

11 Shien-Tai Pan , Kimming So , Joseph T. Rahmeh, Improving the accuracy of dyr using branch correlation, Proceedings of the fifth international conference on Archite programming languages and operating systems, p.76-84, October 12-15, 1992, Bos United States

12 James E. Smith, A study of branch prediction strategies, Proceedings of the 8th Computer Architecture, p.135-148, May 12-14, 1981, Minneapolis, Minnesota, Unite

13 Amitabh Srivastava , Alan Eustace, ATOM: a system for building customized pro Proceedings of the ACM SIGPLAN '94 conference on Programming language design a p.196-205, June 20-24, 1994, Orlando, Florida, United States

14 Tse-Yu Yeh , Yale N. Patt, Two-level adaptive training branch prediction, Procee international symposium on Microarchitecture, p.51-61, September 1991, Albuquerq Rico

15 Tse-Yu Yeh , Yale N. Patt, A comparison of dynamic branch predictors that use t history, Proceedings of the 20th annual international symposium on Computer archit 16-19, 1993, San Diego, California, United States

16 T. Yeh, "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanism Superscalar Processors," Computer Science and Engineering Div. Tech. Report CSE- Michigan, Ann Arbor, MI, Oct. 1993.

17 Cliff Young , Michael D. Smith, Improving the accuracy of static branch predictic correlation, Proceedings of the sixth international conference on Architectural suppor languages and operating systems, p.232-241, October 05-07, 1994, San Jose, Califc

18 C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Corre Prediction," Technical Report 06-95, Center for Research in Computing Technology, Cambridge, MA, Mar. 1995.

↑ **CITINGS** *27*

S. Reches , S. Weiss, Implementation and analysis of path history in dynamic branch Proceedings of the 11th international conference on Supercomputing, p.285-292, Jul Austria

A. N. Eden , T. Mudge, The YAGS branch prediction scheme, Proceedings of the 31st international symposium on Microarchitecture, p.69-77, November 1998, Dallas, Tex

Eric Sprangle , Robert S. Chappell , Mitch Alsup , Yale N. Patt, The agree predictor: a reducing negative branch history interference, ACM SIGARCH Computer Architecture p.284-291, May 1997

Po-Ying Chang , Eric Hao , Yale N. Patt, Alternative implementations of hybrid branch Proceedings of the 28th annual international symposium on Microarchitecture, p.252 29-December 01, 1995, Ann Arbor, Michigan, United States

Sangwook P. Kim , Gary S. Tyson, Analyzing the working set characteristics of branc Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitect 1998, Dallas, Texas, United States

Marius Evers , Sanjay J. Patel , Robert S. Chappell , Yale N. Patt, An analysis of corre

predictability: what makes two-level branch predictors work, ACM SIGARCH Comput v.26 n.3, p.52-61, June 1998

Ravi Nair, Dynamic path-based branch correlation, Proceedings of the 28th annual ir on Microarchitecture, p.15-23, November 29-December 01, 1995, Ann Arbor, Michig

Sanjay Jeram Patel , Marius Evers , Yale N. Patt, Improving trace cache effectivenes and trace packing, ACM SIGARCH Computer Architecture News, v.26 n.3, p.262-271

M. Anton Ertl , David Gregg, Optimizing indirect branch prediction accuracy in virtual ACM SIGPLAN Notices, v.38 n.5, May 2003

Joel Emer , Nikolas Gloy, A language for describing predictors and its application to a ACM SIGARCH Computer Architecture News, v.25 n.2, p.304-314, May 1997

Pierre Michaud , André Seznec , Richard Uhlig, Trading conflict and capacity aliasing predictors, ACM SIGARCH Computer Architecture News, v.25 n.2, p.292-303, May 1

Eitan Federovsky , Meir Feder , Sholomo Weiss, Branch prediction based on universa algorithms, ACM SIGARCH Computer Architecture News, v.26 n.3, p.62-72, June 19

Chih-Chieh Lee , I-Cheng K. Chen , Trevor N. Mudge, The bi-mode branch predictor, annual ACM/IEEE international symposium on Microarchitecture, p.4-13, December (Triangle Park, North Carolina, United States

Nicolas Gloy , Cliff Young , J. Bradley Chen , Michael D. Smith, An analysis of dynam schemes on system workloads, ACM SIGARCH Computer Architecture News, v.24 n.

Dennis C. Lee , Patrick J. Crowley , Jean-Loup Baer , Thomas E. Anderson , Brian N. characteristics of desktop applications on Windows NT, ACM SIGARCH Computer Arc n.3, p.27-38, June 1998

Jared Stark , Marius Evers , Yale N. Patt, Variable length path branch prediction, ACI n.11, p.170-179, Nov. 1998

Jamison D. Collins , Hong Wang , Dean M. Tullsen , Christopher Hughes , Yong-Fong P. Shen, Speculative precomputation: long-range prefetching of delinquent loads, AC Architecture News, v.29 n.2, p.14-25, May 2001

Stuart Sechrest , Chih-Chieh Lee , Trevor Mudge, Correlation and aliasing in dynami SIGARCH Computer Architecture News, v.24 n.2, p.22-32, May 1996

Rastislav Bodík , Rajiv Gupta , Mary Lou Soffa, Interprocedural conditional branch el Notices, v.32 n.5, p.146-158, May 1997

I-Cheng K. Chen , John T. Coffey , Trevor N. Mudge, Analysis of branch prediction vi ACM SIGPLAN Notices, v.31 n.9, p.128-137, Sept. 1996

André Seznec , Stephen Felix , Venkata Krishnan , Yiannakis Sazeides, Design trade conditional branch predictor, ACM SIGARCH Computer Architecture News, v.30 n.2,

James R. Larus, Whole program paths, ACM SIGPLAN Notices, v.34 n.5, p.259-269,

Tao Li , Lizy Kurian John , Anand Sivasubramaniam , N. Vijaykrishnan , Juan Rubio, I improving operating system effects in control flow prediction, ACM SIGPLAN Notices, 2002

Cliff Young , David S. Johnson , Michael D. Smith , David R. Karger, Near-optimal int alignment, ACM SIGPLAN Notices, v.32 n.5, p.183-193, May 1997

Cliff Young , Michael D. Smith, Static correlated branch prediction, ACM Transactions Languages and Systems (TOPLAS), v.21 n.5, p.1028-1075, Sept. 1999

Kevin Skadron , Pritpal S. Ahuja , Margaret Martonosi , Douglas W. Clark, Improving returns with return-address-stack repair mechanisms, Proceedings of the 31st annua international symposium on Microarchitecture, p.259-271, November 1998, Dallas, T

↑ **INDEX TERMS**

Primary Classification:
D. Software
↳ D.3 PROGRAMMING LANGUAGES
↳ D.3.4 Processors
↳ Subjects: Optimization

Additional Classification:
D. Software
↳ D.3 PROGRAMMING LANGUAGES

F. Theory of Computation
↳ F.3 LOGICS AND MEANINGS OF PROGRAMS

General Terms:
Algorithms, Measurement, Performance

↑ **Collaborative Colleagues of:**
Nicolas Gloy:
J. Bradley Chen
Michael D. Smith
Cliff Young
Michael D. Smith:
Sarita V. Adve
Trevor Blackwell
Karl S. Brace
Erik Brynjolfsson
Doug Burger
Brad Calder
Kee Chan

Gang Chen
J. Bradley Chen
Alok N. Choudhary
Antonio Dias
Rudolf Eigenmann
Yasuhiro Endo
Jesse Z. Fang
Catherine H. Gebotys
Nicholas Gloy
Nicolas Gloy
Nikolas Gloy
Nikolas Clemens Gloy
Kim Hazelwood
Glenn Holloway
Mark Horowitz
Mark A. Horowitz
David S. Johnson
Mike Johnson
Mahmut T. Kandemir
David R. Karger
Murali Krishnan
Monica S. Lam
Ruby B. Lee
David J. Lilja
Paul C. Martin
Margaret Martonosi
David Mazières
Pinaki Mazumder
Todd C. Mowry
Alasdair Rawsthorne
Rahul Razdan
Stuart Schechter
Margo Seltzer
Omri Traub
Zheng Wang
Pen-Chung Yew
Cliff Young
Reginald Clifford Young
Xiaolan Zhang

Cliff Young:

J. Bradley Chen
Nicolas Gloy
David S. Johnson
David R. Karger

Michael D. Smith

This Article has als been published in:

- **ACM SIGARCH Computer Architecture News**
  Volume 23 , Issue 2 (May 1995)

A Comparative Analysis of Schemes for Correlated Branch Prediction

Cliff Young, Nicolas Gloy, and Michael D. Smith
Division of Applied Sciences
Harvard University, Cambridge, MA 02138
{cyoung, ng, smith}@das.harvard.edu

Abstract

Modern high-performance architectures require extremely accurate branch prediction to overcome the performance limitations of conditional branches. We present a framework that categorizes branch prediction schemes by the way in which they partition dynamic branches and by the kind of predictor that they use. The framework allows us to compare and contrast branch prediction schemes, and to analyze why they work. We use the framework to show how a static correlated branch prediction scheme increases branch bias and thus improves overall branch prediction accuracy. We also use the framework to identify the fundamental differences between static and dynamic correlated branch prediction schemes. This study shows that there is room to improve the prediction accuracy of existing branch prediction schemes.

Keywords: branch prediction, branch correlation, branch stream characteristics.

1 Introduction

Recent work in branch prediction has led to the development of both hardware and software schemes that achieve good prediction accuracy by exploiting branch correlation [4, 9, 11, 14, 15, 16, 17]. However, little attention has been paid to *why* these schemes behave better than prior ones and to *where* further improvements can be made. In this paper, we describe an analytic framework that helps answer these questions based on the fundamental characteristics of the branch prediction problem. In addition, we use the observations based upon this framework to indicate potentially-fruitful research directions that will allow computer architects to improve branch prediction accuracy. Further improvements in branch prediction accuracy will enhance the effectiveness of global instruction schedulers and the performance of multiple-instruction-issue machines.

Branch prediction addresses two basic problems: predicting the direction of conditional branches, and quickly fetching instructions from the predicted target. These problems can be addressed separately, and in this paper, we limit ourselves to the former. In other words, we consider a *branch prediction scheme* to be a technique for improving performance by anticipating the outcome of conditional branches. Other work has shown how to couple a branch prediction scheme with a branch target buffer to eliminate the performance penalties of branches [7].

Why branch prediction schemes perform differently is just as important as how well they perform. Only after explaining why a scheme works can one understand appropriate ways to improve or alter it. Recent work by McFarling [9] and by Chang et al. [4] uses analysis, reasoning, and experimentation to devise better hardware schemes for correlated branch prediction. In particular, McFarling [9] noticed significant redundancy in the two-level index of the correlation-based branch prediction scheme proposed by Pan, So, and Rahmeh [11]. By hashing the branch history with the branch address, McFarling's *gshare* scheme often improves prediction accuracy under the constraint of a fixed-size table of predictors. Similarly, Chang et al. [4] noticed that, for a fixed-size table of predictors, branches biased to one particular branch direction more than 95% of the time exhibited better prediction accuracies on a two-level adaptive scheme [14] when one decreased the branch history length, while the rest of the branches exhibited better prediction accuracies when one increased the branch history length. This observation led them to propose several new hybrid branch prediction schemes with better overall prediction accuracies.

Still, it is more difficult to understand the actual workings of today's branch prediction schemes than it needs to be. To make it easier to develop optimizations such as those proposed by McFarling [9] and Chang et al. [4], we present a unifying framework that allows one to analyze and categorize branch prediction schemes. Because the framework is based on a theoretical model of the branch prediction problem, it is general enough to encompass all branch prediction schemes proposed to date. The framework focuses attention on how a prediction scheme assigns the dynamic branches of the program to individual predictors. This information then directs our analysis of and our search for weaknesses in a particular scheme, and allows us to isolate and compare different factors that affect prediction accuracy. In particular, we explore the fundamental differences between hardware- and software-based branch prediction schemes that exploit branch correlation. This analysis suggests several ways to improve the overall prediction accuracy of today's branch prediction schemes.

Section 2 describes our framework for classifying and analyzing branch prediction schemes. To demonstrate the generality of our framework, Section 2 presents many of today's popular branch prediction schemes in framework terms. In Section 3, we use the framework to explore the issues in when (and thus why) static schemes for correlated branch prediction work. Section 4 goes on to compare the differences between static and dynamic schemes for correlated branch prediction. As an example of the power of our approach, we also describe changes to correlation-based static and dynamic prediction schemes that improve their overall prediction accuracy. Section 5 summarizes the findings of this work.

2 A Framework for Branch Prediction

Given a conditional branch in a program, the goal of a branch prediction scheme is to predict accurately the outcome of that conditional branch (i.e. that the branch will take or that the branch will fall through).[1] The most accurate branch prediction schemes predict the next action of a branch based on some function of the past actions of that branch and possibly other branches in the program. To understand the capabilities of these branch prediction schemes and to compare competing schemes in a meaningful manner, we must be able to identify and quantify the important properties of branch prediction schemes. To achieve this goal, this section defines a set of mathematical tools that allow us to analyze program and branch behavior in an abstract manner.

2.1 Basic Definitions and Goals

Let a *branch execution* $e = (b, d)$, $e \in \mathbb{Z} \times \{0, 1\}$ be a pair consisting of an identifier $b \in \mathbb{Z}$ and a direction variable $d \in \{0, 1\}$. Intuitively, the identifier uniquely specifies a static branch in a program, and the direction variable indicates the direction that the branch went. We define an *execution stream* or just *stream* as a sequence of branch executions. Intuitively, this corresponds to a branch trace of one invocation of a program, identifying in trace order the conditional branches executed and the directions that they went. A stream can also be formed by concatenating the streams of multiple invocations of a program (possibly with different inputs). We refer to the original stream of all executions in a run of the program as the *program execution stream*. A *substream* of a stream s is a subsequence of s.

A *predictor* is a simple mechanism that predicts the next direction of a stream. A predictor may consider program characteristics (e.g. the opcode of the next branch to predict) in addition to any part of the past program execution stream.[2] The *accuracy* of a predictor is the number of correct predictions divided by the total number of predictions; accuracy measures how closely the predicted stream matches the actual stream.

A *prediction scheme* is a comprehensive mechanism that takes a program execution stream, divides it into substreams, and directs each substream to a unique predictor. Figure 1 illustrates this concept. The objective in dividing the execution stream into substreams is that each substream should be more accurately predictable by its predictor. The accuracy of the prediction scheme is the total number of correct predictions divided by the total number of predictions.

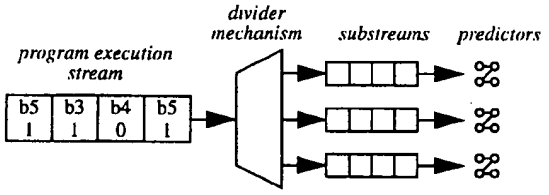


Figure 1. Framework for describing any prediction scheme. The divider mechanism splits the program execution stream into substreams, each of which is predicted by a single predictor.

1 As a point of interest, the goal of a branch prediction scheme is slightly different than the goal of the computer architect. A computer architect's goal is to find a branch prediction scheme that provides the best performance (at possibly the smallest cost), and this may not be the scheme with the best prediction accuracy

2. Here, we mean past program execution stream in the most general sense so that we can consider branch executions from previous runs of the program (as are required for a profile-based predictor)

2.2 Dividing Streams

Based on our formal definition of a prediction scheme, the key to building a more accurate prediction scheme involves the selection of the "right" divider and "good" predictors. In this subsection, we review several current methods for dividing a stream, and we discuss the intuition behind these approaches. Once we have described the important properties of streams that relate to the problem of branch prediction, we then discuss existing predictors and their important characteristics.

Existing schemes divide the program execution stream in a variety of interesting ways. In the simplest case, the divider is the identity function; the program execution stream is fed to a single predictor. The prediction scheme that statically predicts all branches taken [12] and the prediction scheme that uses a single 2-bit saturating up/down counter for all branches [7] are both examples of the identity divider function.

The most popular divider function in today's microprocessors partitions the program execution stream based on the static branch identifier. This partitioning ideally forms one substream for each static branch in the program (a *per-branch substream*) as shown in Figure 2. Formally, if there are n static branches in the program, then the divider creates n substreams, one for each static branch identifier. The divider assigns the ith execution $e_i = (b_i, d_i)$ to the substream that corresponds to b_i. The intuition behind this divider is that each branch should have its own predictor because the characteristics and past history of this branch are a good predictor of its future behavior. Both the per-branch 2-bit counter scheme[3] [7] and per-branch profile-based prediction scheme [10] partition the program execution stream in this manner.

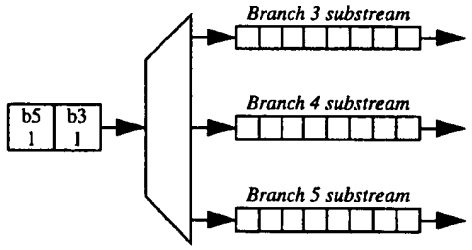


Figure 2. Subdividing the program execution stream into per-branch substreams.

More recent branch prediction schemes further subdivide the per-branch streams. The intuition behind these schemes is that finer decomposition of a per-branch stream can increase the predictability of the individual substreams. For instance, Pan, So, and Rahmeh [11] describe a scheme (which Yeh and Patt call *GAs* [14]) that partitions each per-branch stream based on the pattern of directions of the k preceding branch executions in the program execution stream, as illustrated in Figure 3. The intuition here is that sections of code deal with related information, so tests of one condition are likely to be placed near tests of related conditions. Formally, consider the ith execution in the program execution stream, $e_i = (b_i, d_i)$. The *GAs* scheme considers not just b_i, but also the directions of the k preceding executions d_{i-1}, d_{i-2},...,d_{i-k}. These k bits are called the *pattern history* of preceding branch executions. The k pattern bits are used to further

3 In this subsection, we ignore implementation issues that keep us from obtaining a hardware predictor per static branch. These issues are addressed in Section 2.4 and Section 4. Here, we are concerned only with the ideal intent of the branch prediction scheme

divide a per-branch stream (based on b_i) into 2^k substreams. We refer to these substreams as *per-branch global-pattern streams.*

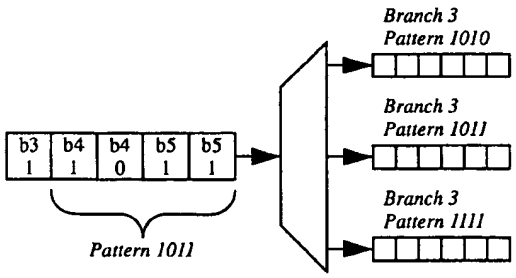


Figure 3. Subdivision in the *GAs* scheme. In addition to branch identifier, the pattern of k preceding branches in the program execution stream is used to further divide the branch streams, so there is one stream per pattern per branch.

As another way to subdivide per-branch streams, Yeh and Patt describe a scheme called *PAs* [15] that uses the last k branches in a per-branch stream to further partition that per-branch stream. This leads to a different set of substreams from the *GAs* scheme. Formally, consider the ith branch execution in the program execution stream, $e_i = (b_i, d_i)$ which is an execution of branch b_i. Let l_1, l_2,...,l_k be the indices of the k previous executions of branch b_i. The *PAs* scheme uses the pattern $d_{l_1}, d_{l_2},...,d_{l_k}$ rather than the pattern $d_{i-1}, d_{i-2},...,d_{i-k}$ to subdivide the per-branch stream (based on b_i) into 2^k substreams. Since the former pattern is determined only by executions of one branch, b_i, *PAs* does not exploit any inter-branch correlation; instead it is designed to exploit repeating patterns in the execution of a single branch. For example, on a loop branch that iterates a constant $c < k$ times, *PAs* approaches 100% branch prediction accuracy, because it will generate substreams consisting solely of a single branch direction (i.e. it can recognize the pattern of c taken branches that will be followed by a fall-through branch). We refer to these substreams as *per-branch branch-pattern streams.*

As a last example of how to subdivide per-branch substreams, we consider our scheme for static correlated branch prediction (*scbp*) [17]. This scheme divides both by branch and by the *path* of branches that led to the executed branch. A path differs from a pattern because it includes both the branch identifiers and the executed directions, not just the concatenation of direction bits. So our static correlated scheme uses the vector

$$(b_{i-1}, d_{i-1}), (b_{i-2}, d_{i-2}), ..., (b_{i-k}, d_{i-k})$$

to encode the path by which b_i was reached, and it uses this vector to subdivide the per-branch stream (based on b_i) into $(2 \times \text{number of static branches})^k$ substreams. We refer to these substreams as *per-branch global-path* streams.

2.3 Predictors and Streams

Under our framework, the divider presents each substream to a single predictor. Each predictor considers some combination of the program characteristics, the past branch execution stream, and its own internal state (if any) in making a branch prediction. In this subsection, we review the range of existing predictors, and we discuss the characteristics of streams that make them predictable.

Predictors can be classified into two major types: static predictors and dynamic predictors. A static predictor must fix its prediction before the program runs, while a dynamic predictor is allowed to change its prediction during program execution. Streams that are largely invariant in branch direction can be accurately predicted by a static predictor. We say that a stream is *strongly biased* if the frequency of one direction is much greater than the frequency of the other direction, and that it is *weakly biased* if the frequencies are close to equal. We refer to the more prevalent direction of the stream as the *majority* direction; the other direction is conversely the *minority.*

Researchers have investigated a variety of static program and branch characteristics to help determine the appropriate static prediction for an execution stream. For example, the simple static branch prediction scheme that always predicts branches to take [12] uses the statistical fact that branches tend to take more often than they fall through. The "backwards taken forwards not taken" (BTFNT) scheme [12] bases the static prediction on the sign of a branch's target offset. Other schemes employ a predictor that computes predictions as a function of the opcode of the branch [7]. Finally, methods like those described by Ball and Larus [2] use sophisticated heuristics about the program structure to generate a static prediction for each branch.

Other than the static characteristics of the program and the branches in the program, researchers use a profile of the dynamic behavior of the program branches, gathered during an earlier program run, to set the static prediction of each branch. If the majority direction remains the same from the profile (training) to the testing run, then a profiled static predictor will perform well. To date, researchers have used only the overall bias of the past branch execution to set the static prediction. In our earlier paper [17], we used other characteristics of the past execution stream, but we used this information to reorganize the program so that its individual branch streams are more strongly biased.

In contrast, dynamic predictors can *adapt* to track the bias of a stream during a single execution of a program. This has the added benefit of not requiring any training or profiling before the program run. Surprisingly, there are very few designs for dynamic predictors. By far, the most popular dynamic predictor is the 2-bit saturating, up/down counter [12]. This predictor forms the basis of all of the correlated branch predictors described by McFarling [9], Pan et al. [11], and Yeh and Patt [14, 15, 16].

Lee and Smith [7] observed that the execution streams of most program branches tend to occur in long runs[4] and that an n-bit counter predictor can exploit this regularity. Smith [12] further observed that a 2-bit counter empirically provides an appropriate amount of *damping* (or hysteresis) to changes in stream direction. A 1-bit counter has no damping (it simply records the direction of the last branch), and 3-bit and higher counters do not appear to offer large cost/benefit advantages over 2-bit counters [12]. Damping trades off adaptability for vulnerability to short minority runs. A 2-bit counter is excellent at predicting streams with long minority runs, and it is damped enough to ignore minority runs of length 1. This allows loop branches, for instance, to incur just one mispredict per loop, instead of two mispredicts (one on loop exit and one on loop reentry).

The distribution of minority run lengths in a stream strongly relates to the effectiveness of today's dynamic predictors. Streams with long runs of one direction followed by long runs of the other direc-

4 A run is a substring of the stream that consists entirely of one direction, and is bounded on either side by executions that go in the opposite direction (or the beginning or end of the stream) Note that a proper substring of a run is not itself a run

tion can be accurately predicted by a dynamic predictor but not by a static predictor. However, a large distribution of short minority runs can cause a dynamic predictor to exhibit worse accuracy than a static predictor because the dynamic predictor adapts too slowly to the changes in the runs.

One other interesting property is the *frequency of recurrent patterns* in a stream. A *pattern* is a non-empty string $w \in \{0, 1\}^*$. A *recurrent* pattern is a substring that occurs multiple times in a stream. Unlike bias and distribution of runs, which are typically used to predict streams that have been divided, this property is exploited by some dividers (e.g. the *PAs* scheme [15]).

2.4 Implementation Details

To this point, our explanations of existing branch prediction schemes focused on the ideal implementation of a scheme. For example, the explanation above describes a per-branch dynamic prediction scheme based on 2-bit counters as able to assign each per-branch stream to a unique 2-bit counter. In actual implementations of per-branch 2-bit counter schemes, this is believed to be impractical. Implementors usually solve this by using just the j least significant bits of the branch address as an index into a table of 2^j counters. This means that, if two conditional branches have the same j lowest bits, their branch streams will be intermingled and sent to a single 2-bit predictor. We call this effect *aliasing*, as the original intent of the 2-bit counter scheme was to provide a single predictor per static branch.

Issues in aliasing have led researchers to develop different branch prediction schemes that we would classify as based on the same ideal branch prediction model. For instance, the *GAs* scheme [14] and McFarling's *gshare* scheme [9] both ideally divide the program execution stream into per-branch global-history substreams, and both use a 2-bit counter as the base predictor. The *gshare* scheme requires fewer 2-bit counters for fixed values of j and k because it exclusive-ors, rather than concatenates, the k bits of pattern history with the j bits of branch address when indexing into the limited table of 2-bit counters. This gives a requirement of $2^{max(k,j)}$ counters, instead of 2^{k+j} counters. Section 4.2 shows that aliasing potentially limits the effectiveness of the ideal divider by intermingling streams that we would ideally like separated.

Static branch prediction schemes that can fix a prediction to each static branch in the program obviously do not suffer from these effects of aliasing. However, static schemes have their own potential limitations due to implementation details. For example, the implementation of our algorithm for static correlated branch prediction [17] does not distinguish between paths that cross a procedure call or return boundary. In other words, they effectively truncate the vector that is used to divide the stream in the cases where a path crosses a procedure boundary. This truncation *merges* streams that would be separated by a more sophisticated divider. We distinguish aliasing from merging: aliasing combines streams from different static branches, while merging combines streams from one static branch.

2.5 Hybrid Approaches

Recent work in branch prediction by McFarling [9] and Chang [4] has proposed hybrid branch prediction schemes which group together multiple basic prediction schemes. The hybrid schemes, either statically or dynamically, select the basic prediction scheme that performs best on a stream. The model in this section can easily be extended to cover hybrid schemes; however, this paper focuses on the power in our model to analyze and improve the individual prediction schemes. Benefits to basic schemes will of course improve the hybrid schemes that include them.

The framework illustrates two distinct avenues of research for improving the accuracy of a branch prediction scheme: one could attempt to improve the sophistication of the ideal model; or one could attempt to remove limitations imposed by current implementation details. The next two sections give examples of each of these approaches, for both static and dynamic branch prediction schemes.

3 Why Static Correlated Prediction Works

The framework described in the previous section gives us a set of terms that can be used to describe, compare, and contrast the behavior of branch prediction schemes. In this section, we examine a simple application of this framework to a pair of similar prediction schemes: per-branch static profile prediction and our static correlated profile prediction [17]. Per-branch static profiling has been shown to work well in a number of studies [5, 10]. In this section, we show how our code transformation exploits branch correlation to increase branch bias.

As noted in Section 2, bias is key to static branch prediction. Figure 4 plots the distribution of taken branch frequency averaged over all benchmarks and data sets. Table 1 presents a summary of our benchmarks and experimental methodology. The "Self History" bars in Figure 4 show that, even for executables produced by today's compilers, most of the dynamic branches are strongly biased. This U-shaped distribution is what makes per-branch static branch prediction effective.

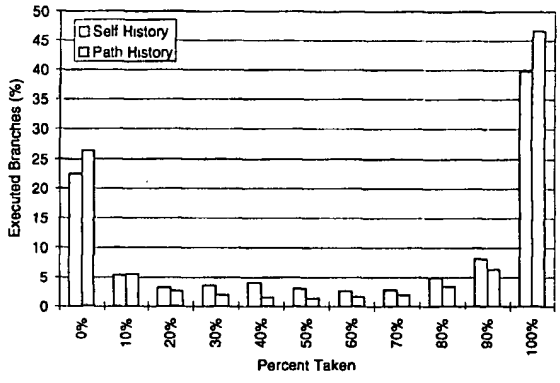


Figure 4. Histogram of branch bias, weighted by execution frequency. This plot averages over all benchmarks, giving equal weight to each data set run. The "Self-History" bars indicate the branch bias in the original executables. The "Path-History" bars indicate the branch bias of executables after transformation to exploit branch correlation with a history depth of 12. The bias values represent the midpoint of a range, e.g. the "10%" bars capture bias values between 5% and 15%. Although this graph averages over all benchmarks and data sets, the trend of increased bias occurred in each individual run. These results train and test on the same dataset.

The effect of exploiting branch correlation is to divide each per-branch stream into several separate streams, discriminating by correlation paths in addition to the static branch identifier. The "Path History" bars in Figure 4 show the distribution of taken branch frequency after our transformation to exploit branch correlation [17]. Compared to the "Self History" bars, the "Path History" bars

	Benchmark and Data Set Descriptions	Total Branches Executed	Static Branches Touched
awk [awk] pattern-directed scanning/processing, GNU ver. 2.15.5			
a	extensive test of awk's capabilities	2.54M	1393
b	simple scanning and printing	0.62M	835
c	generate max array of 3 arrays	4 99M	968
compress [comp]. compression using adaptive Lempel-Ziv, SPECint92			
in	SPECint92 reference input	11 4M	277
jarg	jargon dictionary (1MB of ASCII)	13 1M	280
ps	15-page postscript paper	2.0M	268
diff [diff]: differential file comparator, GNU version 2.6			
a	two C files with 3 diffs	0.43M	646
b	two latex files with many diffs	0.27M	704
xsim	xsim sources with many diffs	0.72M	711
eqntott [eqn] boolean equation to truth table conversion, SPECint92			
fx2fp	8-bit fix to floating point encoder	29.4M	533
tbra	MIPS R2000 taken branch decode	19.3M	528
espresso [esp]: boolean minimization, SPECint92			
bca	SPECint92 reference input	73.9M	1722
cps	SPECint92 reference input	83 1M	1845
ti	SPECint92 reference input	87 4M	1899
grep [grep]: pattern searching program, GNU version 2.0			
a	search for a constant string (2 hits)	0.07M	611
khad	complex regular exp. (100% hits)	0 14M	966
re3	search for a regular exp. (21 hits)	0 33M	878
sc [sc] spreadsheet program, SPECint92			
li	SPECint92 short input	23 5M	1614
lb1	SPECint92 reference input	179 3M	1642
lb3	SPECint92 reference input	44.4M	1538
xlisp [li]: lisp interpreter, SPECint92			
newt	square root via Newton's method	0 11M	550
q4	4 queens problem	0 41M	605
q7	7 queens problem	32.4M	605

Table 1: Benchmark and data set descriptions The results in this paper were derived from trace-driven simulations. We collected the traces using ATOM v1.1 [13]. We compiled the SPECint92 benchmarks using cc version 2.0 0 and the optimization level specified in the SPEC makefiles. The additional benchmarks were compiled using gcc v2.6 0 (-O3). All of the experiments were performed on a DEC 3000/400 running OSF/1 version 2.0

exhibit a larger percentage of strongly biased branches. Over 70% of dynamic branches now occur in streams that are highly predictable. In other words, the more finely subdivided per-branch global-path substreams are more predictable than the coarsely divided per-branch substreams. As we show further in Section 4.3, correlation shifts the distribution of streams and their dynamic branches toward stronger bias.

For static profile prediction to be practical, the static predictions chosen must be valid across invocations of the program. If the majority direction of a stream differs between the profiled (training) data set and the running (testing) data set, then a static predictor will suffer. Fisher and Freudenberger [5] examined a number of different benchmarks and data sets under static profile prediction, and determined that good prediction could be achieved even while training and testing on different data sets. Our experiences so far with various static correlated branch prediction schemes show similar results, although we have not yet done a comprehensive study. An exhaustive treatment of data variance is outside the scope of this paper.

4 Comparing C rrelated Schemes

This section uses the framework to tackle the much harder problem of comparing static and dynamic correlated branch prediction schemes. Superficially, one can compare the prediction accuracy reported by the designers of static and dynamic correlated schemes, but this numerical comparison is unenlightening. For example, in an earlier paper [17], we found that our static correlated branch prediction scheme did not achieve as high a prediction accuracy as the published dynamic correlated schemes. We cannot conclude from these results, however, that the dynamic schemes are necessarily better than static schemes since these schemes differ in more than their base predictors.

Aside from the fundamental differences between a static and a dynamic predictor, our framework suggests that there are three major implementation differences in the divider function: the use of path versus pattern history, the aliasing of multiple (possibly unrelated) branches to the same predictor, and the lack of correlation information across procedure call boundaries. Path history is used in our software prediction scheme, while all current correlated branch prediction schemes based on a hardware table of predictors use pattern history. The aliasing of per-branch streams occurs in hardware-based branch prediction schemes but not in the profiled branch prediction schemes. Finally, our software scheme for correlated branch prediction, unlike the hardware-based schemes, does not exploit correlation across procedure call and return boundaries. Each of these implementation differences can be seen as a limitation that keeps the implemented divider from behaving as precisely as an ideal mathematical divider. Sections 4.1 through 4.3 show, by focusing on *gshare* [9], *GAs* [14], and our static correlated branch prediction scheme (*scbp*) [17], that the removal of these implementation differences can improve the prediction accuracy of correlated branch prediction schemes.

Once we have equalized the divider function, an interesting question to ask is how much benefit one gets from the use of a dynamic predictor in a correlated scheme. Section 4.4 presents one answer to this question by comparing the prediction accuracy of a correlated branch prediction scheme that uses either static predictors or 2-bit dynamic predictors. This experiment uses a theoretical divider function that is uninhibited by the implementation effects of Sections 4.1 through 4.3. Through this experiment, we can begin to understand the true need for dynamic predictors. By understanding where a dynamic predictor is beneficial, we expect to understand how to develop new code transformations to improve the static prediction schemes.

4.1 Paths versus Patterns

As explained in Section 2.2, an implementation difference exists between the divider used in our *scbp* scheme and the one used in an ideal *GAs* scheme. Our *scbp* scheme is based on a per-branch global-path divider that uses a history consisting of a path vector (path history), while *GAs* is based on a per-branch global-pattern divider that uses a history consisting of the pattern of the directions of the most-recent branch executions (pattern history). Path history should provide better correlation information than pattern history, because path history is a superset of pattern history. Path information includes the branches by which the current branch was reached, not just the pattern of directions that they went to reach

the current branch. For example, path information can differentiate between two streams in which two branches take to reach a third block. Figure 5 shows a simple example where that path history achieves a better prediction accuracy than the pattern history.

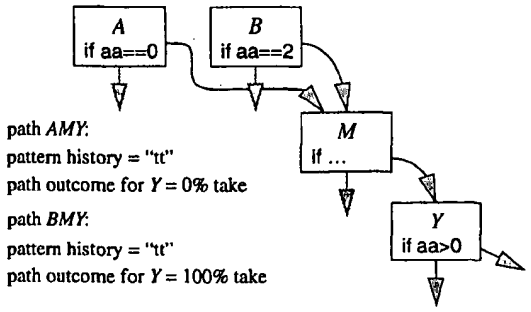


Figure 5. Example illustrating the benefit of path history over pattern history. Path *AMY* and *BMY* are indistinguishable using pattern history, but distinguishable using path history.

To quantify the benefits of path information, we simulated a *scbp* scheme that used only pattern information, and we compared the prediction accuracy of this scheme against the prediction accuracy of the per-branch profile scheme and a *scbp* scheme using path history. These results are summarized in Figure 6. For all benchmarks, the pattern-based *scbp* scheme shows significant improvements over per-branch static profile prediction. There is a small improvement in mispredict rate when path history is used, ranging from negligible in *diff.a* to removing 14% of mispredicted branches in *espresso.bca*. There is a measurable benefit to exploiting path history instead of pattern history, but the majority of advantage is gained just from pattern information.

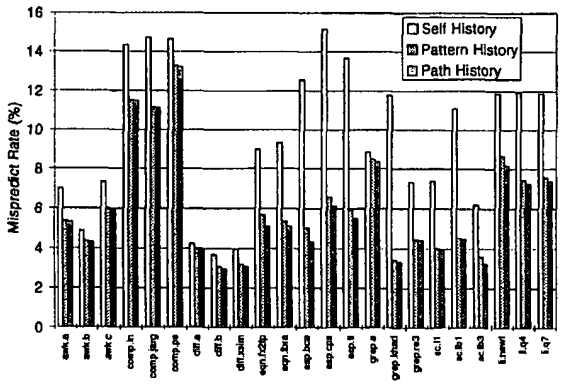


Figure 6. Mispredict rates of per-branch static branch prediction, static correlated branch prediction using pattern history, and static correlated branch prediction using path history. Both correlated schemes use a history depth of $k = 12$. All schemes train and test on the same data set.

From this result in static branch prediction, we would like to generalize to say that all branch prediction schemes could be improved by incorporating path history. But to do this, we need to isolate out the other factors that affect prediction accuracy so that these other factors do not overwhelm the gains due to using path instead of pattern history (and thus cloud the results). We will return to this issue at the end of the next section.

4.2 Aliasing

One important factor that differentiates a static profiled branch prediction scheme from a dynamic branch prediction scheme stems from the fact that dynamic branch prediction schemes map unevenly distributed information like branch address and pattern history into indexed, regular hardware structures (i.e. a table of predictors). In Section 2.4, we defined the term aliasing to describe the situation when, due to implementation limitations, a divider forces streams from different branches to map to the same predictor. A static profiled branch prediction scheme does not suffer from aliasing effects since each branch encodes its predictor function.

Aliasing does not directly imply penalties to prediction accuracy. If two branches with different majority direction alias to the same counter, but one executes 1,000 times followed by the other executing 1,000 times, the loss due to aliasing is negligible. However, if the two branches alternate in trace order, then aliasing may cause significant misprediction. To relate aliasing back to prediction accuracy, we define three kinds of aliasing: If an aliased counter predicts an execution correctly while the corresponding per-stream counter predicts it incorrectly, we call that execution an instance of *constructive* aliasing since the aliasing improves prediction accuracy. Conversely, if the aliased counter mispredicts while a per-stream counter would have predicted correctly, we call that an instance of *destructive* aliasing since the aliasing reduces prediction accuracy. Finally, if the aliased counter predicts an execution correctly (incorrectly) and the corresponding per-stream counter predicts correctly (incorrectly) too, we call that execution an instance of *harmless* aliasing since the aliasing does not change prediction accuracy.

Our intuition is that aliasing is generally bad for prediction accuracy. Since a branch prediction table is a kind of cache, aliasing is analogous to conflict misses in a cache. Instead of suffering conflict misses though, aliased predictors suffer from muddled predictions. As in a cache, increasing the size of the prediction table can help to reduce conflicts (and increase prediction accuracy), as is shown in most of the dynamic branch prediction literature [10, 11, 14, 15, 16]. Chang et al. [4] show benefits to separating out strongly biased branches from weakly biased branches, noting that using static prediction on the strongly biased branches reduces contention (aliasing) in the table of 2-bit counters. Unlike cache conflict misses though, aliasing can be constructive or harmless in addition to destructive. It is important then to understand how aliasing affects the design space of dynamic branch prediction schemes. In particular, we investigate the question of how often aliasing happens in dynamic correlated branch prediction schemes and how this aliasing affects the prediction accuracy.

To see how common aliasing is, we instrumented our hardware simulations to count the number of static branches that map to each 2-bit counter. We examined the usage patterns of the per-branch 2-bit counter scheme, the *GAs* scheme, and the *gshare* scheme for a fixed size table of 4096 2-bit counters. As an example of the type of results we saw, Figure 7 plots the distribution of the number of branches that alias to each counter for each scheme under the *awk.a* benchmark. From Figure 7, one can see that aliasing happens infrequently in the standard 2-bit counter scheme. This makes sense since all benchmarks touch noticeably fewer than 4096 static branches (see Table 1). Aliasing increases in *GAs*, and reaches very high levels under *gshare* since these schemes produce significantly more than 4096 substreams from their dividers. Under *gshare*, running the *espresso* and *sc* benchmarks, aliasing happens so often that in some cases no counters have fewer than three different branches aliased to them. Figure 8 shows detail on constructive versus destructive aliasing in *awk.a* under *gshare*;

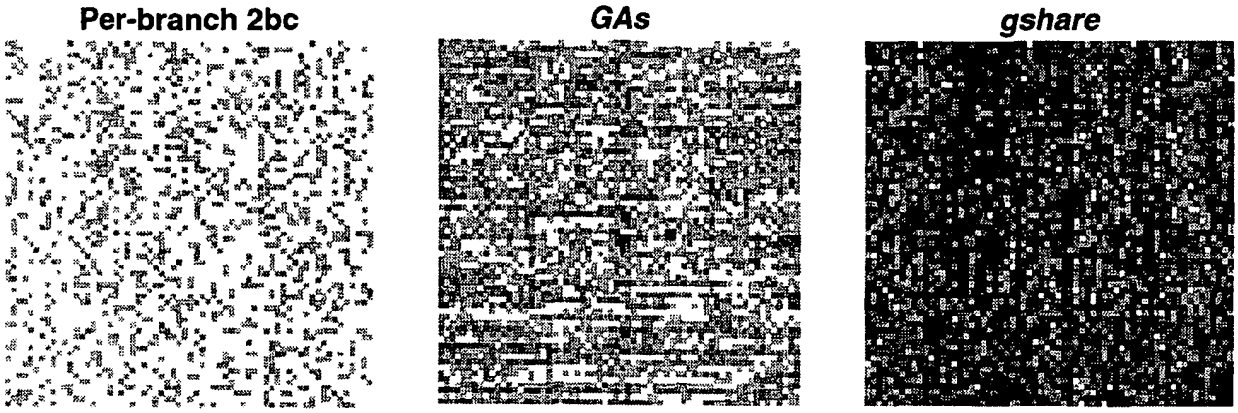


Figure 7. Aliasing in a 4096 counter table for *awk.a* under the per-branch 2-bit counter (12 bits of branch address), *GAs* (6 bits of branch address concatenated with 6 bits of branch history), and *gshare* (12 bits of branch address exclusive-ored with 12 bits of branch history) schemes. White squares represent unused counters; black squares represent counters with seven or more aliased streams.

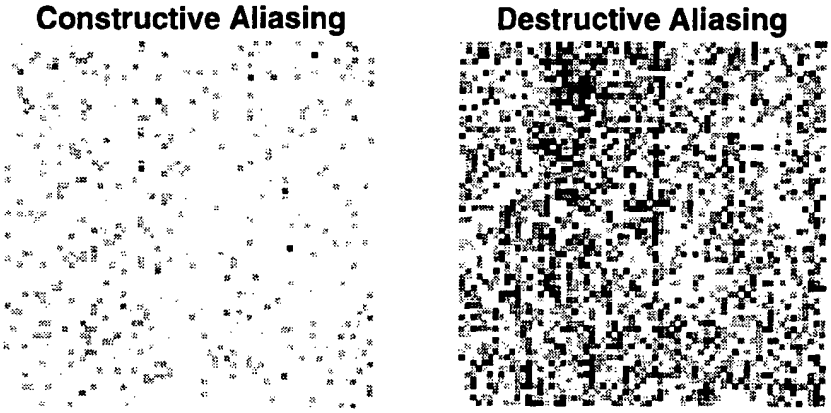


Figure 8. Constructive and destructive aliasing in *awk.a* under the *gshare* scheme. In the left graph, gray scale indicates constructive aliasing, with black representing the maximum attained value of 32. In the right graph, gray scale indicates destructive aliasing, with black representing a difference of 32 or more correct incorrect predictions due to destructive aliasing.

constructive aliasing is both rare and much smaller in magnitude that destructive aliasing.

The more aggressive correlated branch prediction schemes produced more substreams under the assumption that this aggressive subdividing would produce more predictable streams. As shown by the prediction accuracies of the schemes in Figure 9, this decision can lead (though it does not always) to a design with a worse prediction accuracy.

If we remove aliasing from the experiment in Figure 9, an unaliased per-branch global-pattern branch prediction scheme should achieve a higher branch accuracy than either *GAs* or *gshare*. To verify this, we modified our hardware simulation so that each per-branch, global pattern stream was assigned its own counter, then recorded how many executions led to destructive and constructive aliasing. Figure 10 presents the results of this experiment for all benchmarks. Clearly, aliasing happens regularly, and it happens destructively. There is often a significant improvement in the prediction accuracy for removing aliasing effects. Better dynamic prediction schemes are theoretically possible if those schemes can exploit the same pattern and address information as *gshare* without suffering destructive aliasing effects.

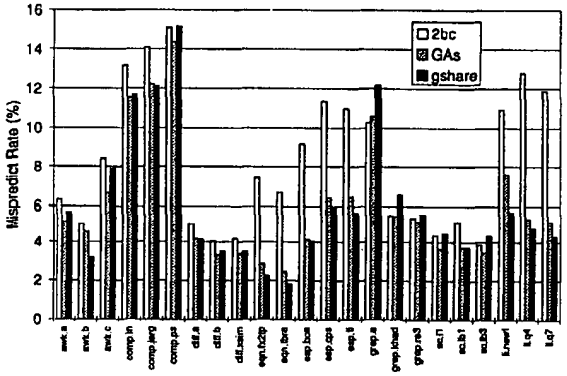


Figure 9. Mispredict rates for the per-branch 2-bit counter (12 bits of branch address), *GAs* (6 bits of branch address concatenated with 6 bits of branch history), and *gshare* (12 bits of branch address exclusive-ored with 12 bits of branch history) schemes; each using a table of 4096 2-bit counters.

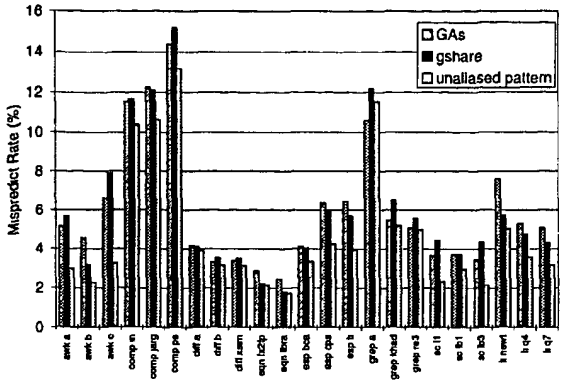


Figure 10. Comparing the mispredict rates of correlated branch prediction schemes that contain aliasing and a branch prediction scheme with a true per-branch, global-pattern divider. All of the schemes use 2-bit counters and a history depth of 12 branches The *GAs* and *gshare* schemes use 4096 counters; the pattern history scheme uses one counter per stream.

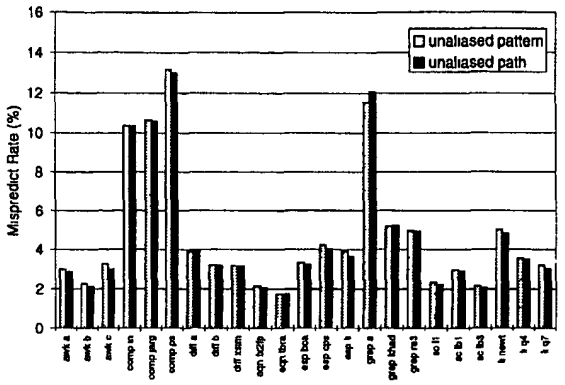


Figure 11. Mispredict rates of schemes using 2-bit counters, a history depth of 12 branches, and a divider without aliasing effects (i.e. one 2-bit counter per stream). "Unaliased pattern" and "unaliased path" depict pattern and path history dividers, respectively.

The *grep.a* bar in Figure 10 is the exception to the trend: the *GAs* scheme shows higher prediction accuracy than the unaliased pattern divider. From Table 1, one can see that *grep.a* executes very few branches. The worse prediction accuracy seems to be a result of the start-up costs of training a 2-bit counter to match a stream's bias. Since the unaliased divider produces more streams than the *GAs* divider, the unaliased divider pays a larger training cost. This larger training cost is significant on short benchmark runs; it might be reduced if schemes that use dynamic predictors could merge streams with similar initial values.

Once we have removed the effects due to aliasing, we are in position to evaluate the benefit of path history over pattern history in dynamic schemes. We extended our simulator to use an unaliased path-history divider with dynamic predictors. The mispredict rates for this path-based predictor are presented in Figure 11. Using paths improves the mispredict rate on the majority of our benchmarks. As in the *grep.a* case from Figure 10, a few of the short benchmarks exhibit worse prediction accuracy under path rather than pattern history due to start-up training costs. Since the magnitude of benefits from a path-based divider are sometimes small, designers must take care that improvements in prediction accuracy due to path history are not swamped by aliasing penalties introduced as part of the modified scheme.

4.3 Cross-Procedure Correlation

So far, the differences we have explored between static and dynamic correlated branch prediction schemes only hurt the prediction accuracy of the dynamic schemes. Yet the overall prediction accuracy of the dynamic schemes is often better. To explain this disparity, we collected statistics of cases where the hardware prediction schemes achieved better per-branch accuracy and then examined the kinds of correlation that occurred. The vast majority of such cases turned out to be cross-procedure correlation: branches that occurred just after a procedure entry or just after a procedure return.

Our *scbp* scheme [17] cannot preserve correlation information across procedure calls. The scheme encodes correlation history into the program counter by duplicating basic blocks. A particular copy of a basic block implies some set of previous execution paths. The problem is that the value of the program counter is effectively reset on a procedure call or return, eliminating correlation information across procedure calls. In terms of the framework, this means that the static scheme's divider is not always capable of using all of the components of the path history vector; the portion of the paths in the vector before a call boundary are merged into a single path.[5] In the extreme, a branch just after a call or return will have no history information available. In contrast, hardware schemes ignore procedure call boundaries, since they record conditional branch directions in additional hardware state.

Some examples of cross-procedure correlation are obvious once they are pointed out:

- The *eqntott* benchmark in the SPECint92 suite uses a quicksort routine to sort bit vectors. A variety of different generic bit-vector comparison functions are passed to *qst()*. Each of these compare routines branches to different return points corresponding to equal, less than, or greater than return values; *qst()* then immediately branches based on the return values. The branch that tests the return value is completely determined by that the branch that set the return value.
- The garbage collector's *mark()* function in the *xlisp* benchmark calls *livecar()* to determine when to follow a node's left sublist. The switch statement inside *livecar()* returns the constant FALSE in many cases; this FALSE return value is then immediately checked by *mark()*.

These kinds of cross-procedure correlation led us to ask how accurately a static prediction scheme could predict if it were possible to preserve path information across procedure boundaries. We modified our trace and simulation environment to record paths across procedure call boundaries, and to simulate the prediction accuracy that would be obtained if a code transformation could preserve all desired correlation information across calls. The prediction accuracy results where we trained and tested on the same data set are summarized in Figure 12. In these results, *compress* shows very little benefit from cross-procedure correlation, but this makes sense because *compress* is implemented as one large loop in a sin-

5 Merging is not always harmful. As part of our *scbp* algorithm, we perform per-branch analysis that intentionally merges path streams with the same majority direction There is no penalty for this kind of merging when using a static predictor, *scbp* exploits this harmless merging to reduce overall code expansion.

gle procedure. In some benchmarks, like *eqntott* and *awk*, more than half of the mispredictions were removed. Other benchmarks showed more modest improvements.

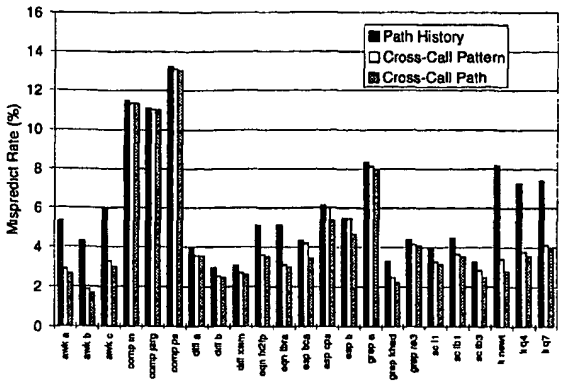


Figure 12. Mispredict rates of scbp using path history (same as the third series in Figure 6) and simulated mispredict rates for cross-call pattern and cross-call path history dividers with static predictors. These results use a history depth of 12 and train and test on the same dataset.

The results in Figure 12 are not necessarily what we would expect from actual implementations of cross-call correlated static schemes, because they train and test on the same data set. This gives best possible static prediction accuracy, rather than what would occur if different training and testing data sets were used. However, these results show that using cross-call correlation we can achieve better static prediction accuracy than was previously believed possible.

Having discussed the implementation differences in dividers, we can now revisit the effect of correlation on bias that we began to explore in Section 3. Figure 13 extends the results shown in Figure 4, adding a new series of columns that shows the bias of streams generated by an unaliased, cross-call, path divider. The improved divider further steepens the U-shaped distribution of bias.

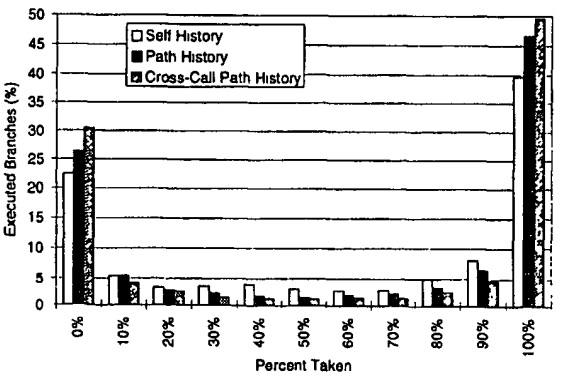


Figure 13. Histogram of branch bias, extending Figure 4. The "Cross-Call Path History" bars show the bias of streams generated by an unaliased, cross-call, path divider. These results use a history depth of 12 and train and test on the same dataset.

Exploiting Cross-Procedure Correlation Statically

We have not yet found a simple code transformation that can generally preserve correlation across calls. However, a number of techniques may be useful: selective inlining [6], template formation, and multiple entry points [1].

Fisher and Freudenberger point out that sophisticated ILP compilers already expect to perform aggressive inlining [5]. Inlining all procedures is impractical, since it is exponential in the depth and degree of the program call graph. But since a small number of procedures make up the majority of program execution cycles [3], it is also likely that a small number of procedures are the best candidates for inlining to extract correlation. The *livecar()* routine in *xlisp* is a great candidate for inlining: it is called in just one place, and it is defined to be local to the *xldmem.c* source file. After inlining *livecar()*, an optimizing compiler could fold the logically correlated branches into a single branch, decreasing the number of static and dynamic branches in the program, and reducing cycle count.

The *eqntott* case, above, is more complicated. Branches in *qst()* correlate into the generic comparison routine that is passed as a function pointer. It is not possible for a compiler to simply inline the comparison routine. However, it would be possible for a compiler or programmer to build different versions of *qst()*, as if *qst()* were a C++ style template function that was instantiated for each comparison function. Since C functions are not first-class types, we could perform function variable propagation analysis to determine all of the possible comparison functions. In fact, in *eqntott*, the comparison functions are constants passed in each call of *qst()*, so we could curry (specialize) the *qst()* call at compile time into a call to the appropriate version of *qst()*.

We can preserve some correlation state across procedure calls by making multiple copies of procedure entry points, one for each relevant past execution history. This allows us to better predict callee branches that correlate back to the caller, but does not help us with the more common case of caller branches that correlate into some utility function.

4.4 Adaptability

The fundamental difference between the static and dynamic correlated schemes is the predictors they use. Dynamic predictors can adapt to track streams during an invocation of the program, while static predictors cannot. This raises the question of whether some streams require the adaptivity of a dynamic predictor to achieve good prediction accuracy. To examine this question, we used the same approach of the previous subsections: subtract out the differences, and see what results. Once again, we used a divider with a path history of length 12 and no aliasing effects. We also made the divider ignore procedure call boundaries like the divider in a hardware implementation.

We classified streams from the divider as "Static Better", "Equal", or "Dynamic Better", depending on whether a static predictor, neither predictor, or a 2-bit counter best predicted the stream. Figure 14 shows the distribution of streams for each benchmark and data set. The "Static Better" bars shows the percentage of streams which were better predicted by a perfectly trained static predictor; the "Static +1" bars show the percentage of streams where the static predictor predicted correctly just one more time than the 2-bit counter. The large number of "Static +1" streams have a majority fall-through direction, and since our simulation initializes 2-bit counters to predict weakly taken, the 2-bit counter incur a mispredict on the first execution in those strongly-biased streams. The

"Dynamic +1" streams are very rare, and there is a small but visible number of "Dynamic Better" streams.

The absolute number of "Dynamic Better" streams is less than 1,000 for all benchmarks except *espresso*. This suggests that there are ways to build better hybrid static/dynamic prediction schemes than that proposed by Chang et al [4]. Their scheme assigns all branches with low bias to dynamic predictors. If we can assign only the rare adaptive streams (which might be aliased together or aliased with statically predictable streams in Chang et al.'s scheme) to their own predictors, while using static predictors for the remaining branches, we should be able to achieve even better prediction accuracy with fewer counters than previous hybrid schemes.

Despite the small percentage of "Dynamic Better" streams in Figure 14, those streams are an important component of overall prediction accuracy. Figure 15 gives details about the *comp.in* bar from Figure 14, plotting the difference in correct predictions. Even though the number of "Dynamic Better" streams is small, the "Dynamic Better" tail is significantly larger than the "Static Better" tail. The integral over the tails gives the differences in correct predictions between schemes using only static predictors and schemes using only dynamic predictors. The "Static Predictor" and "2-bit Counter" bars of Figure 16 compare the mispredict rates of such schemes. Even though we exaggerated the benefit with a static predictor by assuming perfect training, Figure 16 shows that the number of dynamic branches that occur in streams with long runs of the minority branch direction is significant—ignoring them will affect prediction accuracy. However, since the number of static streams requiring an adaptive predictor is very small, the possibility exists for a compiler to selectively apply techniques like predication [8] to these few streams. The vast majority of streams can be handled using simple static branch prediction techniques.

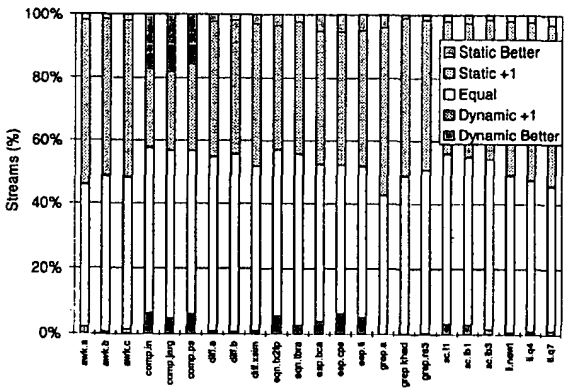


Figure 14. Distribution of streams under an unaliased, cross-call, path divider, depending on whether the streams were predicted better by a perfectly trained static divider or by a 2-bit counter. The "+1" categories contain streams where one of the predictor types correctly predicted just one more execution than the other predictor type. The "Better" categories contains streams where one predictor correctly predicts greater than one more execution than the other predictor.

Hybrid prediction schemes can mix static and dynamic predictors in one scheme. The "Best Predictor per Stream" bars show the mispredict rate as if the best predictor (2-bit counter or static) for a stream was assigned on a per-stream basis, instead of assigning all

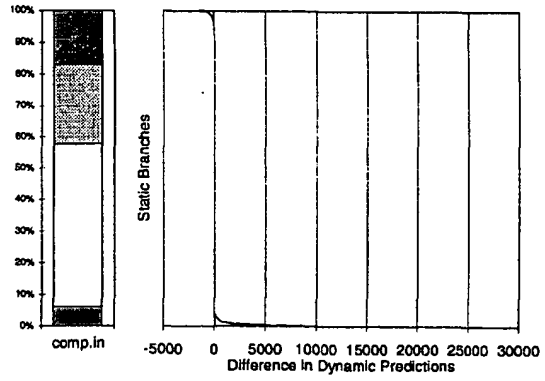


Figure 15. Detailed information on the *comp.in* bar from Figure 14. The horizontal axis shows the difference in correct predictions by the static and 2-bit counter predictors. Positive values correspond to the 2-bit counter predicting more accurately; negative values correspond to the static predictor predicting more accurately.

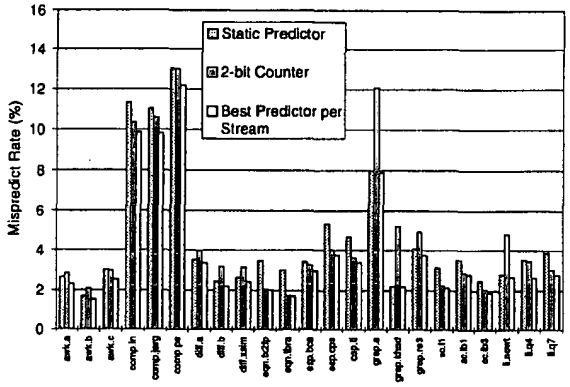


Figure 16. Mispredict rates under an unaliased, cross-call, path divider, comparing assigning all streams to perfectly trained static predictors ("Static Predictor"), all streams to 2-bit counters ("2-bit Counter"), and the best predictor for an entire stream to that stream ("Best Predictor per Stream").

streams to a single kind of divider. These bars show that schemes using a mix of static and dynamic predictors can achieve very high prediction accuracies.

In the long term, adaptability may be the only thing that separates dynamic and static schemes, since static schemes can take cross-call correlation into account, and dynamic schemes can exploit path history and may be able to reduce aliasing problems. Correlation provides a useful tool to reduce the amount of adaptivity (both in dynamic branches and stream distribution) in a program, but no current methods allow us to completely eliminate the need for adaptivity. Hybrid schemes that use the techniques explored in this paper may be able to find efficient ways to separate and handle adaptive streams.

5 Conclusions and Future Work

Before one can build better branch prediction schemes, one must understand how and why existing schemes work. We presented a framework for analyzing and categorizing branch prediction schemes. The framework partitions schemes into two major parts: a *divider* and *predictors*. Dividers attempt to partition the program execution stream into substreams that are individually more predictable than the original stream. All known branch prediction schemes fit into this framework. The framework provided the motivation for all of the studies in this paper, allowing us to practically and systematically analyze the differences between schemes.

Profiled per-branch static branch prediction works because programs have a large percentage of branches that are strongly biased. Correlation changes the distribution of streams to increase the percentage of branches that are strongly biased. Correlation reduces the diversity of branch streams, making profiled static correlated branch prediction more accurate than profiled per-branch static branch prediction.

Under our framework, state-of-the-art static and dynamic prediction schemes differ in four major qualities: use of pattern versus path history, aliasing effects, ability to exploit cross-procedure correlation, and adaptivity.

- Path history is slightly better than pattern history in exploiting branch correlation.
- Correlated dynamic branch prediction schemes utilize more 2-bit counters in their tables, but simultaneously increase the amount of aliasing. Removing the effect of aliasing increases prediction accuracy, suggesting that work should be done to limit aliasing in dynamic branch prediction schemes.
- Cross procedure correlation limits the accuracy of static branch prediction schemes. We showed some large potential benefits to cross-procedure correlation in static schemes. We are pursuing several practical techniques that allow static schemes to exploit cross procedure correlation.
- The percentage of adaptive streams is small, but that the dynamic branches executed in adaptive streams are significant.

We have not reached the limits of existing basic branch prediction schemes. We have demonstrated potential for increased prediction accuracy in each of the areas above. Dynamic branch prediction schemes will benefit from methods to control aliasing and to exploit path history. Static branch prediction schemes will benefit from techniques that exploit cross-procedure correlation and reduce the need for adaptive predictors.

6 Acknowledgments

We thank Hewlett-Packard and Digital Equipment Corporation for their generous donation of several HP 9000 Series 700 and DECstation 3000 Series workstations on which we ran our tracing and analysis tools. We would also like to thank Brad Calder for his useful comments on early versions of this paper. Cliff Young is funded by a Graduate Fellowship from the Office of Naval Research. Michael D. Smith is supported by a National Science Foundation Young Investigator award, grant number CCR-9457779.

An expanded set of the results for this paper can be found in [18].

7 References

[1] V. Bala, personal communication, Oct. 1994.

[2] T. Ball and J. Larus, "Branch Prediction for Free," *Proc. ACM SIGPLAN 1993 Conf. on Prog. Lang. Design and Implementation*, Jun. 1993.

[3] J. Bentley, *Programming Pearls*, Addison-Wesley, 1986.

[4] P. Chang, E. Hao, T. Yeh, and Y. Patt, "Branch Classification: a New Mechanism for Improving Branch Predictor Performance," in *Proc. 27th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, November 1994.

[5] J. Fisher and S. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[6] W. Hwu and P. Chang, "Inlining Function Expansion for Compiling C Programs," *Proc. ACM SIGPLAN 1989 Conf. on Prog. Lang. Design and Implementation*, Jun. 1989.

[7] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, 17(1), Jan. 1984.

[8] S. Mahlke, et al., "Characterizing the Impact of Predicated Execution on Branch Prediction," *Proc. 27th Annual Intl. Symp. on Microarchitecture*, Nov. 1994.

[9] S. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, June 1993.

[10] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," *Proc. of 13th Annual Intl. Symp. on Computer Architecture*, Jun. 1986.

[11] S. Pan, K. So, and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, Oct. 1992.

[12] J. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Annual Intl. Symp. on Computer Architecture*, Jun. 1981.

[13] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. SIGPLAN '94 Conf. on Prog. Lang. Design and Implementation*, Jun. 1994.

[14] T. Yeh and Y. Patt, "Two-Level Adaptive Branch Prediction," *Proc. 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*, Nov. 1991.

[15] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," *Proc. 20th Annual Intl. Symp. on Computer Architecture*, May 1993.

[16] T. Yeh, "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," Computer Science and Engineering Div. Tech. Report CSE-TR-182-93, Univ. of Michigan, Ann Arbor, MI, Oct. 1993.

[17] C. Young and M. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," *Proc. 6th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, October 1994.

[18] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," Technical Report 06-95, Center for Research in Computing Technology, Harvard University, Cambridge, MA, Mar. 1995.